View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Electronic Archive of Kyiv Polytechnic Institute

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»
Факультет інформатики та обчислювальної техніки
Кафедра автоматики та управління в технічних системах

«До захисту допущено»
Завідувач кафедри
_____ О.І. Ролік
«___»_____2019 р.

Дипломний проект

на здобуття ступеня бакалавра

з напряму підготовки 6.050103 «Програмна інженерія»

на тему: «Веб-застосунок для пошуку репетитора на базі мікросервісної архітектури»

Виконав:
студент IV курсу, групи ІТ-51
Свириденко Олександр Андрійович _____

Керівник:
Професор кафедри АУТС, ДТН, доцент Корнієнко Б.Я. _____

Рецензент:
Доцент кафедри ОТ, КТН, доцент Павлов В.Г. _____

Засвідчую, що у цьому дипломному проекті немає запозичень з праць інших авторів без відповідних посилань.
Студент _____

Київ – 2019 рік

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Факультет інформатики та обчислювальної техніки
Кафедра автоматики та управління в технічних системах

Рівень вищої освіти – перший (бакалаврський)
Напрям підготовки – 6.050103 «Програмна інженерія»

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ О.І. Ролік
«___»_____2019 р.

ЗАВДАННЯ
на дипломний проект студенту
Свириденку Олександру Андрійовичу

1. Тема проекту «Веб-застосунок для пошуку репетитора на базі мікросервісної архітектури», керівник проекту професор Корнієнко Богдан Ярославович, затверджені наказом по університету від «___»_____ 2019 р. №_____

2. Термін подання студентом проекту _____

3. Вихідні дані до проекту ___*.NET Core, Entity framework Core, Microsoft SQL Server, Postgre SQL, Redis, Docker, RabbitMQ*____ __

4. Зміст пояснювальної записки __*Вступ. 1. Аналіз предметної області. 2. Розробка та виділення мікросервісів. 3. Проектування архітектури та серверної частин. 4. Обгрунтування вибору засобів реалізації 5. Реалізація проекту. Висновки*_
5. Перелік графічного матеріалу (із зазначенням обов'язкових креслеників, плакатів, презентацій тощо)_____ *1. Діаграма використання – плакат. 2. Діаграма послідовності – плакат. 3. Діаграма діяльності– плакат. 4. Діаграма бази даних–плакат*
7. Дата видачі завдання _____

Календарний план

№ з/п	Назва етапів виконання дипломного проекту	Термін виконання етапів проекту	Примітка
1	Отримання завдання	25.02.2019	
2	Збір інформації	05.03.2019	
3	Вивчення та вибір варіанту для розробки архітектури застосунку	12.03.2019	
4	Створення та моделювання сутностей для баз даних мікросервісів	23.03.2019	
5	Розробка серверної частини	28.03.2019	
6	Розробка взаємозв’язків між компонентами системи	18.04.2019	
7	Розробка інтерфейсу користувача та тестування взаємодії компонентів	27.04.2019	
8	Оформлення дипломної роботи	31.05.2019	
9	Отримання допуску до захисту	04.06.2019	
10	Захист дипломної роботи	20.06.2019	

Студент О.А. Свириденко

Керівник проекту Б.Я. Корнієнко

АНОТАЦІЯ

Свириденко О.А. Веб-застосунок для пошуку репетитора на базі мікросервісної архітектури. КПІ ім. Ігоря Сікорського, Київ, 2019.

Проект містить 66 сторінки тексту, 11 рисунків, 7 таблиць, посилання та 20 літературних джерел, додатки.

Ключові слова: наукова праця, мікросервісна архітектура, розподілені системи, контейнеризація, веб-інтерфейс.

Об'єктом дослідження є процес пошуку користувачами спеціалістів для індивідуального навчання.

Мета роботи – розробка системи для пошуку репетиторів з віддаленим доступом через веб-інтерфейс та використанням сучасних підходів у розробці архітектури.

У дипломному проекті вирішується проблема з створення застосунку для пошуку спеціалістів для індивідуального навчання. Проект відповідає найсучаснішим вимогам для веб-застосунків, надає можливості шукати та спілкуватись з репетиторами, встановлювати час навчання та інше. Використано сучасні підходи в розробці та забезпечено високий рівень швидкодії.

Результати дипломної роботи можуть бути застосовані для використання її у цілях бізнесу та запуск застосунку для використання в реальних умовах.

SUMMARY

Sviridenko O.A. Web application for searching tutor based on micro-server architecture. KPI them Igor Sikorsky, Kyiv, 2019.

The project contains 66 pages of text, 11 figures, 7 tables, links and 20 literary sources, applications.

Key words: scientific work, microservice architecture, distributed systems, containerization, web interface.

The object of the research is the process of finding users of specialists for individual training.

The purpose of the work is to develop a system for the search of tutors with remote access through the web interface and using modern approaches in the development of architecture.

In the diploma project it is decided to create an application for the search for specialists for individual training. The project meets the most up-to-date web application requirements, provides the ability to search and communicate with tutors, set up training time, and more. Modern approaches in design are used and high level of performance is ensured.

The thesis work can be applied to its use for business purposes and the launch of the application for use in real-world conditions.

Ном. поз.	Формат	Позначення	Найменування	Кількість аркушів	№. екз.	Примітки
1			Документація загальна			
2						
3			Знову розроблена			
4						
5						
6						
7	А4	ІТ51.220БАК.002 ПЗ	Пояснювальна записка	66		
8						
9	А3	ІТ51.220БАК.002 Д1	Діаграма компонентів	1		
10						
11	А3	ІТ51.220БАК.002 Д2	Діаграма послідовності	1		
12						
13	А3	ІТ51.220БАК.002 Д3	Діаграма використання	1		
14						
15	А3	ІТ51.220БАК.002 Д4	ER – діаграма	1		
16						
17						
18						

					ІТ51.220БАК.002 ПЗ		
Змн.	Арк.	№ докум.	Підпис	Дат			
Розроб.		Свириденко О.			*Веб-застосунок для пошуку репетитора на базі мікросервісної архітектури*	Літ.	Арк. / Аркушів
Перевір.							
Реценз.						*НТУУ "КПІ" ім. Ігоря Сікорського ІТ-51*	
Н. Контр.							
Затверд.							

Пояснювальна записка
до дипломного проекту
на тему: «Веб-застосунок для пошуку репетитора на базі мікросервісної архітектури»

Київ – 2019 рік

ЗМІСТ

ВСТУП

Система освіти постійно розвивається і з часом необхідність у підготовці до різних контрольних та тестових завдань тільки зростає. Вступні іспити, ЗНО, ДПА, TOEFL та інші екзамени потребують хорошої підготовки для отримання задовільного результату. Державні навчальні заклади не завжди можуть дати достатній рівень знань. В такому випадку можна звернутись до спеціалістів для індивідуального навчання, які професійно готують учнів до іспитів.

Метою даного проекту є створення застосунку для пошуку спеціалістів для індивідуального навчання. Обрано реалізацію через веб-застосунок з використанням сучасних підходів до розробки. Веб-застосунок дозволить користувачам незалежно від платформи або пристрою мати доступ до системи та даних. Проект повинен відповідати найсучаснішим вимогам для веб-застосунків, надавати можливості шукати та спілкуватись з репетиторами, встановлювати час навчання та інше. Для можливості використання сучасних підходів в розробці проекту та забезпечення високого рівня швидкодії потрібно обрати сучасні технології для створення інтерфейсу користувача та роботи з даними.

Предметом дослідження є застосунки для пошуку та комунікації між користувачами з певною спеціалізацією, а саме пошуком спеціалістів для індивідуального навчання.

Практичною цінністю розробки такої системи є подальша можливість використання її у цілях бізнесу та запуск застосунку для використання в реальних умовах.

1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1.Огляд аналогів.

Зараз існує багато застосунків з подібною функціональністю. Деякі аналоги є міжнародними платформами, деякі позиціонують себе українські платформи. Частина має вузьку специфікацію – наприклад, вивчення іноземних мов.

Прикладом такої платформи є preply.com. Вона спеціалізується на можливості знаходження спеціаліста для індивідуального вивчення мов, хоча варто зауважити що на сайті представлена можливість знайти спеціаліста для вивчення мови програмування або, наприклад, грі на саксофоні. Сайт має сучасний та інтуїтивно зрозумілий інтерфейс, декілька мов інтерфейсу, що дозволяє працювати йому на міжнародному ринку, зручну систему відображення зайнятості репетиторів, відгуків про них. Інтегрована система оплати дозволяє мінімізувати можливості для недоброчесної поведінки та гарантувати отримання коштів та послуг сторонами.

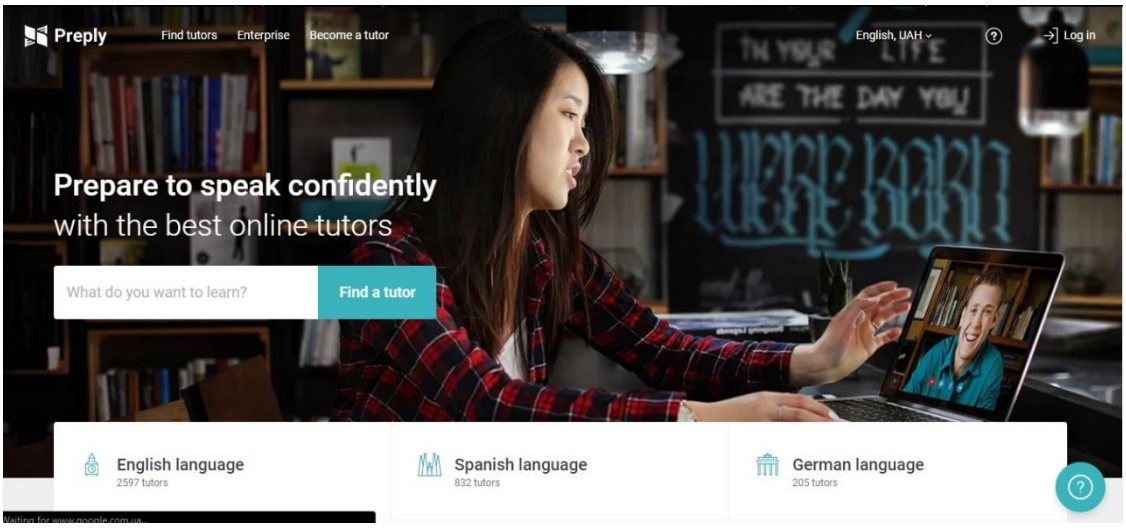

Рисунок 2.1 – Інтерфейс головної сторінки сайту preply.com

З технічної сторони можна виявити, за допомого аналізу заголовків запитів, що серверну частину створено за допомогою фреймворка Django (мова програмування – Python), а фронтенд частину – за допомогою бібліотеки React (Javascript).

Прикладом платформи, орієнованої на внутрішній ринок України може служити buki.com.ua. Цей сайт дозволяє знайти спеціаліста в багатьох сферах, не тільки шкільної та університетської програми. Сучасний інтерфейс з лаконічним викладенням інформації, який зображено на рисунку 2.2. Зручна система фільтрів для пошуку саме того спеціаліста, який потрібен. На відміну від попереднього немає зручної, з точки зору користувача, форми, де можна побачити коли репетитор готовий займатися з учнем, для отримання цієї інформації потрібно зв'язатись. Хоча сайт і орієнтований на українську аудиторію, інтерфейс доступний у 3 мовах – українська, російська та англійська.

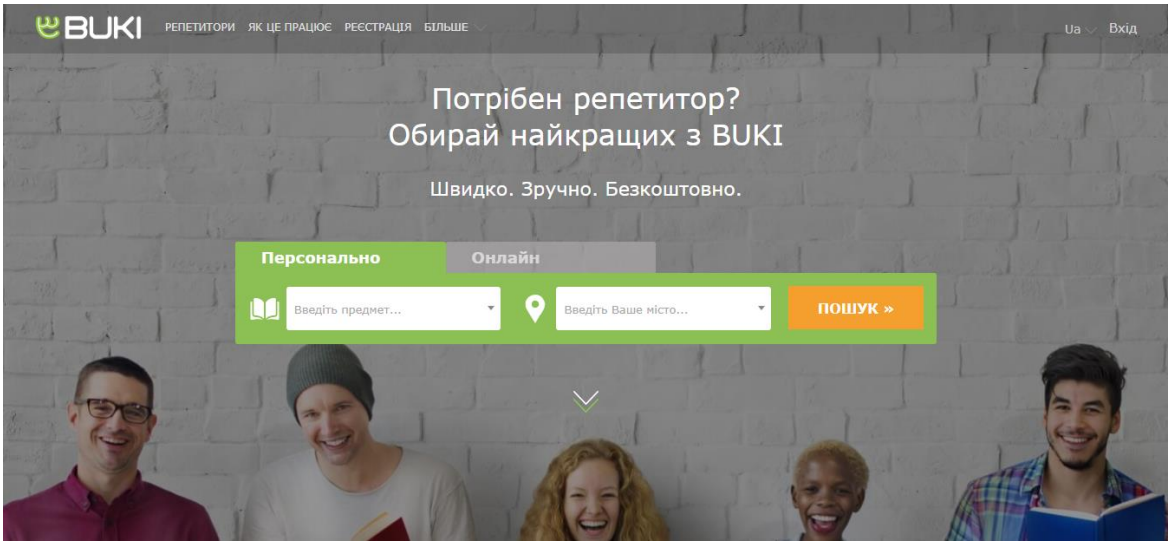

Рисунок 2.2 – Інтерфейс головної сторінки сайту buki.com.ua

Фронтенд частина написана з використанням бібліотеки jQuery, бекенд частину важко проаналізувати, оскільки заголовків з інформацією серед запитів немає.

Ще одним прикладом системи з подібними функціями може слугувати 100balov.info. Цей сайт позиціонує себе як саму популярну платформу для пошуку репетиторів на всій території України та по Skype. Загалом інтерфейс сайту помітно застарілий, хоча вся необхідна інформація присутня. Важливим недоліком відносно попередніх прикладів є відстуність можливості змінити мову інтерфейсу, тому сайт можливо використовувати тільки в російській версії.

Сайт розроблено за допомогою мови програмування PHP, фронтенд частина написана з використанням бібліотеки jQuery.

Окремо можна виділити «локальні» сайти для пошуку репетитора, які дозволють користувачу знайти спеціаліста або пропонувати свої послугу в одному місті. Приклад – сайт kiev.repetitors.info. Сайт має широку спеціалізацію, можна знайти репетитора з різних дисциплін. Варто зауважити, що сайт є частиною платформи, яка дозволяє шукати репетиторів у різних містах та областях України. В інтерфейсі доступно перемикання з російської мови на українську. Загалом дизайн сайту можна оцінити як дещо застарілий.

1.2. Актуальність

Проаналізувавши дані, отримані при огляді аналогів, можна дійти висновку, що поточні проекти, які реалізують подібний функціонал мають проблеми. Більша частина цих сайтів давно не оновлювала дизайн, що призвело до невідповідності їх інтерфейсу сучасним нормам. Оскільки зараз система освіти розвивається, все більше і більше людей потребують допомоги спеціалістів для індивідуального навчання для успішної здачі екзаменів, ЗНО або вивчення іноземних мов. Тому в цьому сегменті ринку потреба у системах з подібною функціональністю буде зростати.

Система з якісним і інтуїтивно зрозумілим інтерфейсом та високою швидкодією з легкістю зможе обійти застарілі аналоги в популярності серед

користувачів. Сучасні підходи в проектуванні та розробці забезпечать платформу можливістю подальшого розвитку системи. Серверна частина з можливостями горизонтального масштабування дозволить системі витримувати велику кількість користувачів.

1.3.Діаграма діяльності

При проектуванні застосунків важливо чітко розуміти не тільки функціональність системи, а і послідовність дій користувачів під час використання системи. Для задоволення цих потреб застосовано діаграму діяльності. Всі основні дії, які користувач може виконувати в рамках системи, описано в діаграмі діяльності, зображеній в додатку В. З неї видно, що користувач може увійти до системи, або зареєструватись в ній при необхідності. Для пошуку спеціалістів можливо використати різні підходи, такі як пошук по локації, пошук по направленню або інші. При необхідності користувач може комбінувати ці підходи для оптимального пошуку. Користувач може створювати запити до репетиторів, змінювати графіки та додавати документи, які пов’язані з навчальним процесом. Також користувач може надсилати користувачам повідомлення, а також редагувати їх.

Висновки до розділу

В даному розділі проведено аналіз предметної області, оглянуто та проаналізовано аналоги. Визначено актуальність системи та оглянуто діаграму діяльності, що необхідна для глибшого розуміння особливостей розробки системи.

2. РОЗРОБКА ТА ВИДІЛЕННЯ МІКРОСЕРВІСІВ

2.1. Визначення мікросервісів та границь їхньої функціональності

При розробці мікросервісної архітектури та кожного мікросервісу зокрема потрібно приділити особливу увагу визначенню мікросервісів, їх властивостей та розмірів. Якщо мікросервіс завеликий – підтримка та розширення функціоналу в ньому стає складним завданням, затратами часу та інших ресурсів порівнянними з додаванням подібного функціоналу в моноліт. Однак якщо мікросервіси є занадто малими то починаються проблеми зовсім іншого типу, а саме – при кожному запиті потрібно виконати велику кількість міжсервісних запитів, які забирають досить багато часу. До того ж складність у логіці та зв'язках між об'єктними сутностями переноситься на міжсервісні зв'язки, що зовсім не полегшує розробку, а інколи і взагалі ускладнює через складності відлагодження асинхронності та міжсервісних зв'язків. Якщо обрано правильний розмір мікросервісів, але неправильно визначено границі функціоналу – проблеми з'являться в необхідності створення великої кількості додаткових запитів до API мікросервісу.

Перш за все – правило, яким потрібно користуватись при визначенні границь функціональності каже, що в кожному з мікросервісів повинен бути створений функціонал лише для розробки однієї можливості, яку очікує бізнес. Тому кожна бізнес можливість системи має бути виділена в окремий мікросервіс.

При прямому «переносі» бізнес можливостей на мікросервіс границі функціональності мікросервісу залежать від бажань бізнесу. Однак не завжди доцільно для розробки нової можливості системи створювати новий мікросервіс. Бувають ситуації, у яких, з точки зору технічної реалізації системи, частину функціоналу в одній бізнес можливості варто перенести до уже існуючого мікросервісу, або створити додатковий мікросервіс. Хоча такі дії і порушують закон Конвея, але більш важливими та вагомими факторами

при розробці мікросервісної архітектури є чіткі границі кожного з сервісів та їх незалежність. При створенні мікросервісів без урахування технічної реалізації, а покладаючись тільки на закон Конвея[6], ми отримаємо систему, яка буде дуже точно відображати систему комунікації між відділами з точки зору бізнесу, однак мікросервіси будуть побудовані з сильною зв’язаністю та порушеннями основних правил розділення на частини. Однак при визначенні мікросервісної структури та границь тільки за технічними правилами ми отримаємо легко підтримувану і логічну систему, яка не відповідає реальним зв’язкам в комунікаціях між відділами. Це в подальшому може призвести до складнощів у розробці нового функціоналу, оскільки після отримання специфікації потрібно витратити час на розподілення роботи між частинами команди, які підтримують сервіси. Така ситуація виникає по причині того, що специфікації для розширення можливостей системи ідуть від певного відділу і стосуються їх сфер дій. В системі з мікросервісною архітектурою побудованою за вищезгаданими правилами модулі і їх функціональність не відповідають цим сферам діяльності.

2.2. Розділення компоненту на мікросервіси

Простим і наглядним прикладом може виступати мікросервіс сповіщення користувачів. Схожий функціонал буде створено в розроблюваній системі для спілкування між користувачами та репетиторами. Бізнес ставить завдання створити можливість розсилання користувачам системи повідомлень та спілкуванню з конкретним користувачем в рамках системи без використання сторонніх програм та сервісів. Цей функціонал варто створити в окремому мікросервісі, оскільки надсилання сповіщень користувачам може виявитись необхідним в різних частинах системи. Візьмемо тут і в подальшому як приклад розроблювану систему пошуку спеціаліста для індивідуального навчання. Система може розсилати користувачам повідомлення з новинами та

оновленнями, користувачі можуть спілкуватися між собою, а репетитор може надіслати користувачу матеріали для уроку або перенести його на більш зручний час. Весь цей функціонал має бути зібраний в один мікросервіс, який буде відповідати за надсилання користувачам будь яких повідомлень.

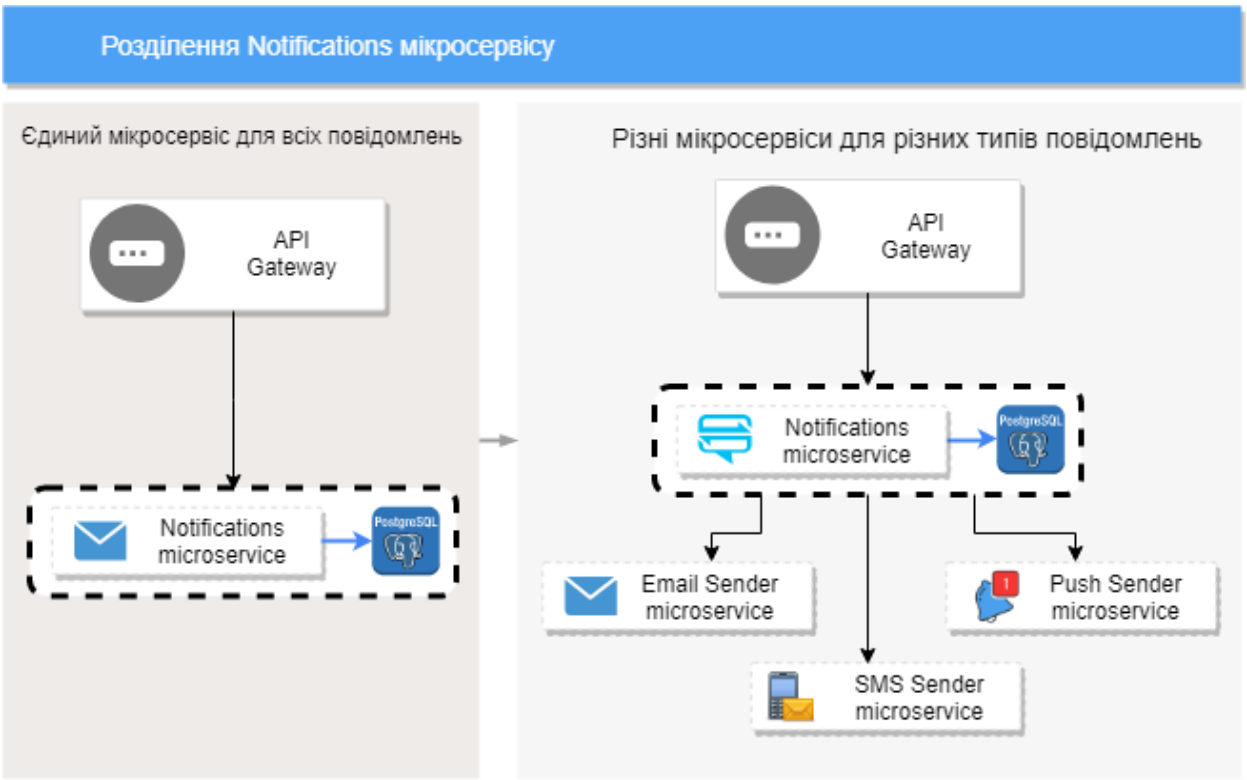

Рисунок 2.1 – Розділення компоненту на мікросервіси

Також можна створити 4 сервіси які будуть обслуговувати систему надсилання повідомлень користувачам. Тоді запит на відправлення повідомлення буде потрапляти до мікросервісу надсилання повідомлень, а уже в цьому мікросервісі буде обрано який саме тип повідомлення потрібно відправити (push, email, SMS чи якийсь інший), а далі запит буде зроблено до відповідного сервісу, який вже буде надсилати повідомлення до користувача обраним ним способом.

Але система тільки в процесі створення прототипу, тому на початковому етапі не має сенсу розбивати сервіс надсилання повідомлень користувачам на

менші сервіси. При необхідності це можна буде зробити у майбутньому. Для полегшення можливої роботи над розбиттям одного мікросервісу на три, або більше, потрібно при розробці дотримуватись

З технічної точки зору це правильний та логічний шлях для вирішення проблеми надсилання користувачам повідомлень, однак зі сторони бізнесу завдання на додавання функціональності може з’явитися з різних частин системи, в технічному плані яким відповідають різні мікросервіси. Це призведе в кінцевому рахунку до необхідності внесення змін у різних частинах проекту і різних мікросервісах. Однак це вимушений вибір оптимального варіанту, тому що розробляти окремий мікросервіс для надсилання повідомлень користувачам до кожного модуля, що потребує подібного функціоналу порушує правило «не повторювати себе» [13], яке стосується коду.

До того ж, при правильному підході необхідність вносити зміни до декількох мікросервісів можна мінімізувати, розробивши логіку таким чином, щоб мікросервіс надсилання займався виключно надсиланням повідомлень користувачам і ніяк не впливав на формування повідомлень. В такому разі зміни в нього потрібно буде вносити тільки у раз необхідності додавання нового типу повідомлень (наприклад, повідомлення в месенджерах), розбитті поточного мікросервісу на декілька інших або змін у вже розроблених частинах.

2.3.Виділення мікросервісу

Процес додавання сервісу до системи очевидний, а процес розбиття уже розробленого сервісу виконується наступним чином[7]. Для прикладу вибрано схему розділення одного з мікросервісів в проекті, а саме Tutors microservice.

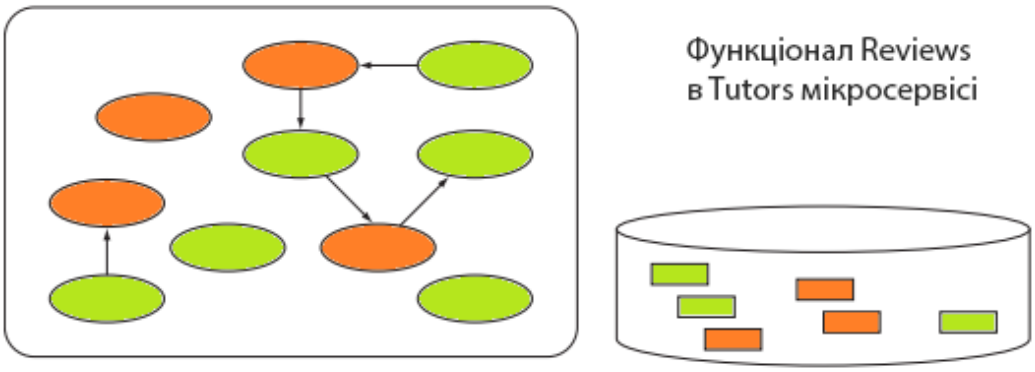

Рисунок 2.1 – Крок 0 для розділенні мікросервісів

На початковому етапі, як можемо побачити з рисунку 2.1, вся функціональність зібрана в одному сервісі. Потрібно визначити, які з функцій поточного сервісу варто винести до окремого мікросервісу. Після визначення цих функцій варто детально проаналізувати та вирішити, які з цих функцій пов’язані з поточним сервісом та побудувати зв’язки між ними. Це потрібно для того, щоб в подальшому розуміти, які зв’язки потрібно буде побудувати між сервісами та інтерфейси, які потрібно буде одразу створити.

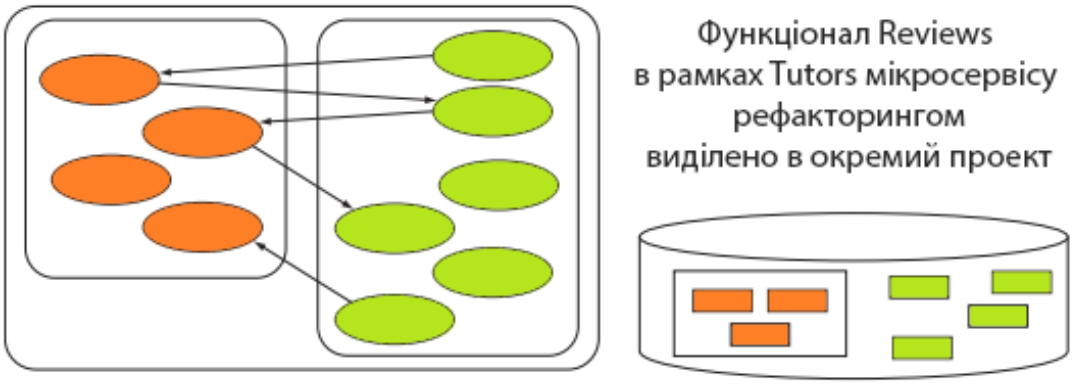

Рисунок 2.2 – Крок 1 для розділенні мікросервісів

Наступним кроком, який візуалізовано на рисунку 2.2, потрібно в розбити поточний сервіс на 2 проекти. З точки зору архітектури це і досі функціонує

як єдиний сервіс, але в структурі коді це вже окремі проекти, які тісно пов’язані між собою функціоналом.

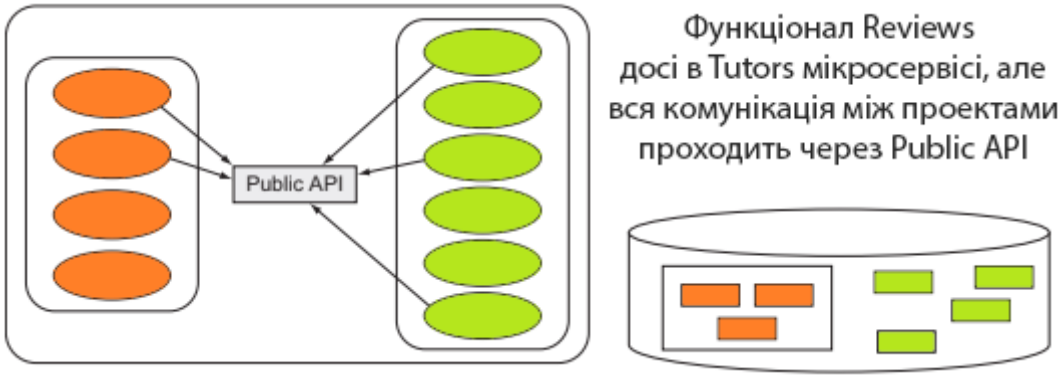

Рисунок 2.3 – Крок 2 для розділенні мікросервісів

Далі потрібно створити API для спілкування між сервісами. Це допоможе розбити тісні зв’язки між ними і перенести ці зв’язки до окремої частини мікросервісу. Це є шаблоном для побудови інтерфейсу спілкування між сервісами в наступному кроці. Результат цього кроку можна побачити на рисунку 2.3.

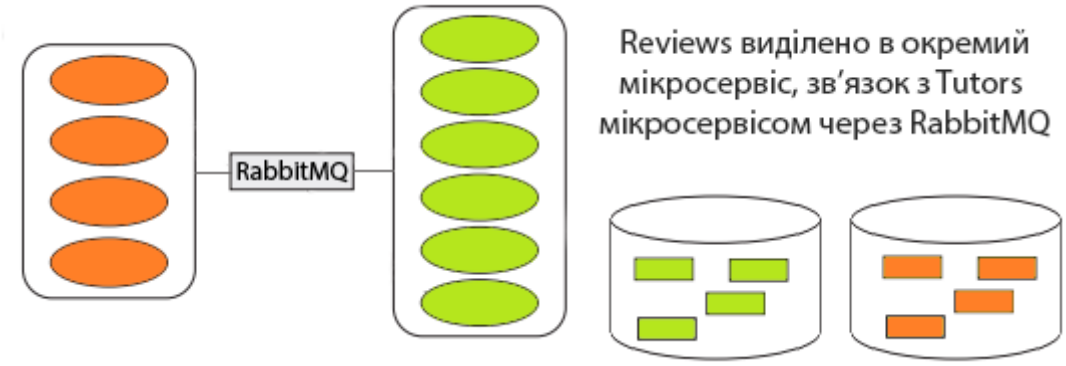

Рисунок 2.4 – Крок 3 для розділенні мікросервісів

Виділений проект потрібно винести як окремий сервіс. На даному етапі з точки зору архітектури ми отримуємо два мікросервіси замість одного, функціональність якого буде розбита згідно з логікою бізнесу та/або технічної

необхідності. Для проектування інтерфейсу комунікації між створеними мікросервісами використовується API з попереднього етапу.

Кінцевий результат розділення мікросервісу візуалізовано на рисунку 2.4.

Висновки до розділу

В рамках даного розділу було розглянуто методи роботи з мікросервісами, розділення мікросервісів та визначення границь їх функціональності.

3. ПРОЕКТУВАННЯ АРХІТЕКТУРИ ТА СЕРВЕРНОЇ ЧАСТИНИ

Найбільш популярними зараз є 2 підходи до побудови архітектури веб-застосунку – це монолітна та мікросервісна. Для пошуку більш підходящої архітектури розглянемо обидві та порівняємо їх плюси і мінуси. Варто зауважити що розділення на 2 основні архітектурні стилі – не зовсім правильне, оскільки існує багато архітектурних патернів, їх визначення є недостатньо чіткими, а реальна архітектура деяких систем знаходиться в нечітких граничних областях на перетині різних архітектур. Також існують системи, архітектура яких не входить ні в першу, ні в другу категорію.

3.1.Монолітна архітектура

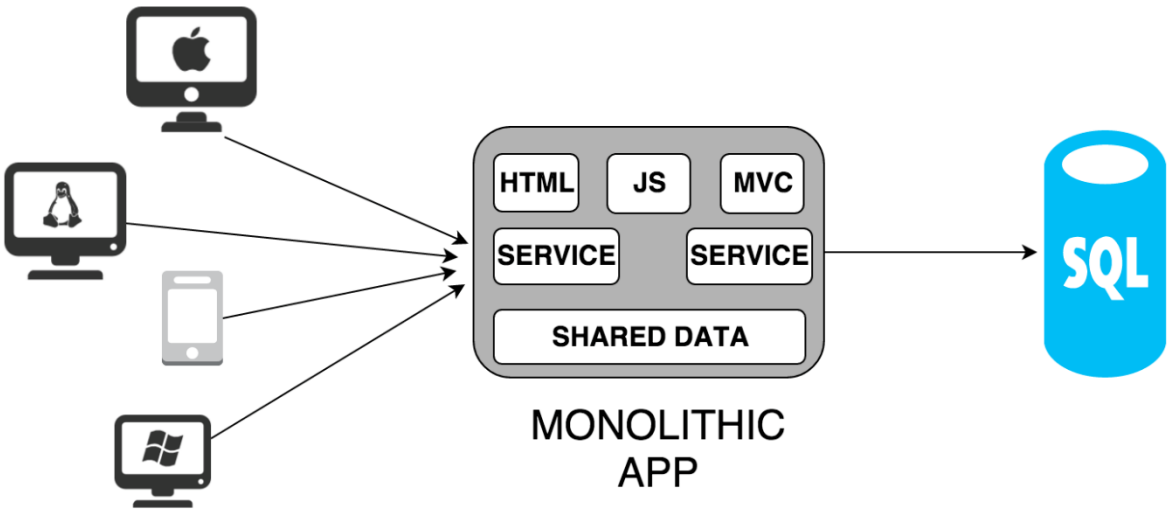

Рисунок 3.1 – Приклад монолітної архітектури [8]

Під поняттям «монолітна архітектура» розуміють те, що застосунок буде виступати в єдиному великому модулі, який має спільний доступ до ресурсів та пам’яті, а всі компоненти тісно пов’язані один з одним. Прикладом побудови такої архітектури може служити схема компоненті на рисунку 3.1.

Основними перевагами такого підходу є можливість простого використання одного і того ж функціоналу у всіх компонентах застосунку (наприклад логування або права доступу). Також, додатковою перевагою є продуктивність, оскільки застосунок з монолітною архітектурою використовує спільну пам’ять, що є швидшим методом комунікації ніж, наприклад, обмін даними між різними сервісами.

До недоліків монолітної архітектури можна віднести сильну зв’язаність між компонентами застосунку, що в подальшу перетворюється у проблеми під час масштабування та додавання нового функціоналу. Також, при великих розмірах системи моноліт є більш складним для розуміння, оскільки рамки компонентів стають розмитими та з’являються неочевидні зв’язки між ними.

3.2.Мікросервісна архітектура

Більш детально розглянемо мікросервісну архітектуру, її сильні та слабкі сторони.

Мікросервісна архітектура несе в собі багато переваг перед монолітною, однак вона включає і серйозні недоліки, вивчивши які легко побачити, що мікросервіси не є доцільними у всіх без виключення системах.

3.2.1. Модульна структура мікросервісів

При правильній розробці архітектури мікросервіси – це чітка модульна структура, яка дозволяє полегшити розробку у великих командах. Така структура досягається за допомогою чітких границь модулів та їх функціональності.

Важливо зауважити, що в при побудові проекту з використанням монолітно архітектури розробка програмних модулів, що є повністю відокремленими є надзвичайно корисним, оскільки це полегшує роботу та

розширення функціональності системи. Якщо потрібно вносити зміни до існуючого функціоналу – то потрібно змінити лише невелику частину, до того ж полегшується робота і рефакторингом коду, оскільки він розбитий на невеликі логічні частини. Однак при рості системи та її функціональності мікросервісна структура по суті стає необхідністю.

На підтримку розробки системи використовуючи такий архітектурний підхід одразу можна навести закон Конвея (Conways Law) [6] – "організації що проектують системи ... змушені створювати проекти які є копіями комунікаційних структур цих організацій." Отож, згідно цього закону, мікросервісна архітектура може повністю відповідати системі комунікації у організації, в якій (або для якої) вона розроблялась.

Монолітна архітектура також може мати добре структуровану модульну систему, яку легко підтримувати та доповнювати. Однак з практичного досвіду більшість розробників знає, що більшість монолітних систем є нагромадженнях коду з складними і часто неочевидними залежностями, різними підходами та стилями розробки. Рефакторинг таких систем перетворюється на величезну проблему, яка потребує великих затрат ресурсів розробників та команд тестування. Відокремленість модулів і їх чіткі границі зберігаються за рахунок між модульних посилань, які в свою чергу дозволяють чітко відокремити доступ до даних функціоналу з одного модуля в іншому модулі. В монолітній архітектурі такого бар’єру не існує, тому функціональність дуже легко додається без врахування модульності, що , в кінцевому рахунку, руйнує модульну структуру і сповільнює швидкість роботи розробників системи.

Іншою не менш важливою перевагою мікросервісної архітектури є децентралізованість даних. Кожен окремий модуль (мікросервіс) використовує власну БД, яка є відокремленої від всіх інших. При розробці модуля також можна використовувати різні БД в залежності від потреби модулів і необхідних технічних характеристик тієї чи іншої бази даних.

Доступ до внутрішніх даних в БД кожного окремого модуля регулюється API цього самого модуля. Такий підхід дозволяє захистити систему від інтеграційних баз даних, що породжують масу проблем в монолітних системах через так звані “шкідливі зв’язки”[9].

Однак така перевага легко може перетворитись у недолік, якщо при проектуванні системи не були сповна враховані всі необхідності бізнесу або окремі модулі системи мають занадто широкі можливості та виконують частину функції, які по логіці повинні були би бути створені у інших модулях. Тоді мікросервісна структура також стане “заплутаною”, API деяких модулів доведеться розширити, а логіку визначення границь мікросервісів буду порушено.

Через вищезгадані обставини часто спочатку розробляють монолітну систему, щоб краще зрозуміти весь обсяг функціоналу, або, як варіант, розробляють систему з розділенням на декілька великих модулів з нечіткими границями. Такі архітектуру не можна назвати модульною. Однак це дозволяє дуже швидко (порівняно з розробкою мікросервісної архітектури) створити робочий проект та вийти на ринок з готовим функціоналом продукту, що є важливою перевагою для бізнесу. Мати готову систему з важкою масштабованістю і не настілки якісно спроектованою архітектурою набагато краще ніж мати систему з якісно розроблено архітектурою та легким процесом масштабування, однак непотрібну зі сторони бізнесу та користувачів. Загалом з точки зору бізнесу розробка системи з мікросервісної архітектурою є ризикованою справою, оскільки для створення такої системи необхідно витратити набагато більше коштів, але ніхто не може гарантувати що ця система стане популярною і зможе окупити кошти, які були витрачені на її розробку. Це стає чи самим основним аргументом на користь розробки системи на базі монолітної архітектури з дотриманням модульності, границь API та способів збереження даних, а з часом готову систему можна розвивати в бік мікросервісної архітектури, виділяти все більше і більше окремих

модулів, які будуть виконувати тільки невелику частину функціоналу та дозволять децентралізувати частину даних. При виділенні модулів з уже робочого моноліту або з більшого модуля розробники мають набагато менше можливостей помилитися і неправильно обрисувати границі, функціонал уже є розділеним і потреби бізнесу уже є зрозумілими та частково розробленими.

Окрема користь мікросервісів проявляться, як уже було вказано раніше, при розробці таких систем у великих командах, що дозволяє легко налагодити взаємодію між частинами команди та вести швидку і ефективну розробку великих частин функціоналу повністю незалежно. При необхідності пришвидшення розробки та розширення функціональності досить легко підключити додаткових розробників. Їм не потрібно буде досліджувати та глибоко розуміти підходи в розробці усієї системи, єдине що потрібно буде – розібратися з внутрішньою побудовою окремого модуля. Це дозволить при необхідності пришвидши будь який процес та етап в розробці системи.

3.2.2. Розподіленість мікросервісної архітектури

Розподіленість дозволяє набагато краще підтримувати модульний підхід, однак вона сама в собі несе серйозні недоліки. При виборі такого підходу одразу виникає велика кількість проблем, які потребують професійного підходу для якісного та швидкого їх вирішення[10].

Враховуючи сучасний розвиток апаратних частин цифрової техніки, якщо бути точним – комп’ютерних процесорів, досить складно написати якісний код , який буде повільно виконуватись. Однак при використанні модульного підходу затримки можуть виникати при виклику інших модулів системи, і якщо один з модулів викликає одразу п’ять інших – затримки при виконанні такого запиту можуть стати критичними. Звичайно, завжди є можливість побудувати асинхронні виклики та зекономити час на очікування паралельністю виконання, а обробляти запитуванні дані по мірі їх отримання,

але асинхронність завжди має ряд складнощів при розробці: вона вимагає високої кваліфікації, досить важко відловлювати помилки та займатись налагодженням. Існують різні методи та підходи для пришвидшення швидкодії системи при комунікації між мікросервісами, але проблема все одно залишається.

Важливим моментом є надійність системи. При розробці монолітної архітектури очікується, що всі внутрішні функції будуть правильно працювати. На противагу цьому все більше і більше розширення функцій системи несе в собі розширення кількості використовуваних мікросервісів, що, в свою чергу, додає можливих проблемних частин у системі, оскільки будь–який запит до іншого сервісу може не спрацювати. Саме через ці проблеми при розробці мікросервісів варто враховувати всі можливі ризики та проблеми.

Варто зазначити що розробка системи з монолітною архітектурою не провидять до повної відсутності таких проблем. Чим більше система розростається – тим більше з’являється необхідність в використанні сторонніх сервісів, комунікація з якими проходить через мережу. Це провидять до таких самих проблем, з якими розробники стикаються при створенні мікросервісної системи. Тому часто команди розробників великих систем більш схильні до переходу від монолітної архітектури до мікросервісної, оскільки в кінцевому рахунку мікросервіси є більш вигідним рішенням не дивлячись на наявні проблеми.

3.2.3. Неузгодженість даних між мікросервісами

Ця проблема постає із децентралізації управління даними в мікросервісних системах. В монолітній архітектурі можливо обновити велику кількість об’єктів за рахунок однієї транзакції, і у випадку виникнення проблем або помилок в даних, які в подальшому можуть привести до помилок у бізнес

логіці, будь яку транзакцію можна повернути і її початкову точку. На противагу монолітній архітектурі у мікросервісній помилки можуть виникнути тільки у деяких об’єктах і транзакції повернуться у початковий стан тільки у частині мікросервісів, а у іншій дані успішно будуть змінені. В такому випадку бізнес логіка система буде працювати з неузгодженими даними, що в подальшому може привести до ще більших проблем та накопичення помилок в даних кожного окремого мікросервісу. При розробці систем з модульною структурою варто пам’ятати про високу вірогідність виникнення таких проблем та створити можливості для визначення моментів неузгодженості даних[11].

Для монолітних систем такі проблем теж не є рідкістю. Яскравим прикладом цього може служити додавання кешування або збереження одного і того ж об’єкту різними користувачами. Кеш може бути застарілим і уже не актуальнім, тому варто створити системи перевірки актуальності кешованих даних. В свою чергу зміни в об’єкті різними користувачами виконуються базуючись на знаннях про початковий об’єкт, і якщо всі зміни будуть збережені то можливе виникнення серйозних проблем неузгодженості та помилок в даних, що в подальшому може призвести до проблем у бізнес логіці.

Отже, монолітні системи також мають проблеми з узгодженістю даних, але в порівнянні з мікросервісними системами їх набагато простіше вирішити, особливо якщо це невелика система.

3.2.4. Незалежне розгортання мікросервісів

Один із основних принципів мікросервісної архітектури – незалежне розгортання. Головна концепція цього принципу – кожен сервіс може бути самостійно розгорнутим на сервері та не бути залежним від інших частин системи і сервісів. Вносячи зміни в один з мікросервісів можна бути певним що це не нашкодить всій системі і дозволить коректно працювати навіть при

наявності помилок у одному з модулів. Як було наведено раніше, при розробці системи з мікросервісною архітектурою варто враховувати можливості відмов сервісів, а це дозволить всій системі коректно працювати навіть при повній відмові одного з сервісів, хоча вона і буде мати не повну функціональність та проблеми з швидкодією.

Незалежне розгортання дозволяє інтегрувати в систему неперервної доставки (Continuous Delivery)[12], що дозволяє постійно розгортати на сервері, або декількох серверах, обновлені сервіси з новим функціоналом, і відразу переходити ді їх тестування. Та в цьому є і невелика складність у розробці мікросервісної системи – для коректної і швидкої розробки просто таки необхідно налаштувати систему неперервної доставки на сервер, оскільки оновлювати вручну кожен сервіс після кожної функції або виправлення помилки – дуже складна та об’ємна робота. Такі можливості є також вигідними і для бізнесу – швидка розробка та доставка на production середовище дозволить швидше за конкурентів додавати функціонал та змінювати систему згідно з попитом ринку.

Варто відмітити, що монолітну систему також можна розгортати в автоматичному режимі, найбільш відомий приклад – Facebook, а не дивлячись на те, що неперервна доставка є одним з основних принципів мікросервісів, при неправильній розробці системи, на перший погляд зручна система автоматизованої доставки викликає багато помилок та вимушеної зміни налаштувань через неправильно побудовані взаємодію між сервісами.

3.2.5. Технологічні можливості мікросервісів

Важливою перевагою мікросервівсів над монолітом є можливість використання різних технологій та мов програмування. Як уже пояснювалося раніше, кожен модуль – це окрема система, яка розгортається незалежно від інших та з своєю системою налаштувань. Часто при розробці великих систем

різні команди займаються різними модулями, що дозволяє проаналізувати функціональність різних модулів та обрати оптимальний набір технологій. Це дозволяє розробити більш гнучку систему, кожен окремий модуль якої оптимально адаптований під виконання поставлених задач.

Також під час розробки великих систем, які підтримуються уже деякий час команда розробників може стикатися з задачами, які вносять конфлікти до системи. Це виражається у проблемах з використанням старих версій бібліотек. В новому функціоналі, який бажають додати до системи, використовуються частини нової бібліотеки, а в уже розробленому та протестованому – стару версію цієї ж бібліотеки. Саме в такі моменти при розробці проекту на монолітній архітектурі виникають проблеми, і командні потрібно вирішити – або не використовувати нову версію бібліотеки, або переписати компоненти та модулі системи, що використовують стару версію.

Мікросервісна архітектура з легкістю справляється з такою задачею, оскільки кожен з сервісів є повністю відокремленою частиною, тому різні модулі можуть використовувати різні версії бібліотеки не маючи при цьому ніяких конфліктів. Можлива ситуація що в одному і тому ж сервісі потрібно використати різні версії бібліотеки і команда знову зтикається з тими ж, здавалося б, проблемами, що і при розробці моноліту. Однак тут варто врахувати модульну структуру і те, що сервіси є невеликими, тому переписати не велику частину коду сервісу, з яким команда давно працює набагато швидше і простіше, ніж вносити зміни у частини системи, з якими команда взагалі не працювала, при цьому маючи всі супутні проблеми, які виникають при додаванні функціоналу та рефакторингу великого монолітного проекту.

3.2.6. Безпека в мікросервісній архітектурі

Мікросервісна архітектура дозволяє без особливих проблем відділити особливо важливу інформацію від інших модулів та зберігати її у окремому

сервісі, в якому розробити складну систему безпеки, що убезпечить дані користувачів від зловмисників. Всі дані, які передаються між мікросервісами можна зашифрувати або використовувати нестандартні протоколи передачі, це також дозволить захистити дані користувачів.

Безпека певних частин системи може стати причиною використання мікросервісів – адже при такій структурі архітектури системи важливі дані користувача є максимально захищеними. До того ж, при розробці монолітних систем часто використовують окремий сервіс для збереження важливих даних, так само як і в мікросервісах розроблять серйозну систему захисту саме для цього сервісу.

Варто підсумувати все вище сказане. Вибір архітектури системи – це складний процес. Перш за все через нечіткість визначень “монолітна архітектура” та “мікросервісна архітектура”. Також існує можливість створити систему, яка не буду вписуватися в формальні рамки ані тої, ані іншої архітектурної моделі. Наступним важливим чинником в виборі є зрілість системи. Якщо система тільки проектується і бізнес не можу бути впевненим в успішній реалізації своїх задумів створення мікросервісної архітектури може бути недоцільним через вищі затрати на розробку, розгортання та підтримку. Не варто забувати і те, що правильний вибір архітектури ніяк не зможе покращити якість коду, написаного розробниками, та злагодженої роботи і взаємодії в команді. Ці фактори зіграють набагато більшу роль, в кінцевому результаті, ніж побудова архітектури на мікросервісах чи на моноліті. На технічному рівні важливо більш зосередитись на розробці чистого коду, написанню тестів та розвитку поточної архітектури системи

Висновки до розділу

В даному розділі було розглянуто різні сучасні підходи до проектування архітектури веб застосунків. Проведено порівняння монолітної та мікросервісної архутектури.

Проведено обширний розгляд особливостей використання мікросервісів у розробці архітектури веб-застосунків, проаналізовано сильні та слабкі сторони.

Обгрунтовано вибір мікросервісної архітектури для розробки проекту.

4. ОБГРУНТУВАННЯ ВИБОРУ ЗАСОБІВ РЕАЛІЗАЦІЇ

На вибір технологій та їх різноманіття впливає використання мікросервісної архітектури та децентралізованість сховища даних. Це дозволяє обрати оптимальний набір технологій для кожного мікросервісу, що в свою чергу підвищує швидкодію та можливості у подальшому ефективно та швидко створювати новий функціонал або вносити зміни до існуючого. Обґрунтовуючи вибір технологій визначальними факторами будуть їх технічні характеристики. Але варто врахувати певні обмеження через необхідність самостійної розробки системи та невелику кількість мікросервісів.

4.1.Серверна частина та мікросервіси

Для розробки мікросервісів у серверній частини використано .NET Core 2.2. Ця технологія є вільним та відкритим фреймворком для розробки веб застосунків. Вона розроблена компанією Майкрософт разом з вільними розробниками і є продовженням лінії технологій .NET Core, яка прийшла на заміну .NET Framework і є повністю переписаним фреймворком, який обєднує раніше окремо створений функціонал ASP.NET MVC та ASP.NET Web API у єдину програмувальну модель. Окрім швидкодії та оптимізації виконання коду, важливою перевагою над .NET Framework є можливість запускати проекти, написані з використанням даної технології, окрім Windows ще й на macOS та Linux. Також проекти побудовані на базі фреймворку .NET Core можуть запускатися на одній машині з використанням різних версій фреймворку, що не було можливим раніше. Остання версія .NET Core – 2.2, вона вийшла 12 квітня 2018 року [14].

Окрім вищезгаданих переваг варто перечислити інші можливості фреймворку:

- розробка без компіляції (компіляція неперервна, отже розробник не повинен додатково використовувати команду компіляції);
- модульна структура розподіляється як NuGet пакети;
- оптимізована для використання в хмарних серверах;
- хост–агностик за допомогою відкритого веб–інтерфейсу для .NET (OWIN) підтримки – працює в IIS або в автономному режимі;
- система створення конфігурації середовища на основі хмар;
- легкий і модульний обробник HTTP запитів;
- відкрите джерело та орієнтоване на спільноту.

Entity Framework 6 використовується як фреймворк для об'єктно–реляційного відображення для ADO.NET. По аналогії з .NET Core поточна версія відрізняється від попередніх, перш за все тим, що тепер Entity Framework 6 відділена від .NET Core.

EF Core – це більш сучасний, легкий і розширюваний варіант Entity Framework, який має дуже схожі можливості та переваги для EF6 [15]. EF Core є повноцінним переписанням і містить багато нових функцій, які не доступні в EF6, хоча й досі не вистачає деяких найсучасніших можливостей відображення EF6. Розгляньте можливість використання EF Core у нових програмах, якщо набір функцій відповідає вашим вимогам. Порівняти EF Core & EF6 розглядає цей вибір більш детально.

Як ORM, EF Core знижує невідповідність між реляційними та об'єктно–орієнтованими світами, дозволяючи розробникам писати програми, які взаємодіють з даними, що зберігаються в реляційних базах даних, використовуючи об'єкти .NET, що представляють домен програми, і виключаючи необхідність для великої частини доступу до даних "сантехнічний" код, який вони зазвичай повинні написати.

Як EF Core, так і EF6 реалізує багато популярних функцій ORM:

- відображення (mapping) даних з бази на обєкти застосунку;
- неперервне відстеження змін;
- використання різних видів завантаження зв’язаних данних[5]:
  - лінивий,
  - явний,
  - неявний;
- використання LINQ to Database для отримання даних від бази;
- використання підходу Code-first;
- візуальний проектувальник моделей для таблиць;
- можливості використання в застосунках на базі .NET Core і .NET Framework;
- можливості підключення та використання різноманітних баз даних з використанням дрйверів, доступні бази – SQL Server, Oracle, MySQL, SQLite, PostgreSQL, тощо;
- різноманітні можливості для відображення даних та створення зв’язків між ними, серед яких:
  - побудова відносин між таблицями один-до-одного, один-до-багатьох та багато-до-багатьох;
  - збереження процедур;
  - успадкування (таблиця за ієрархією, типом або іншою таблицею);
  - складні типи;

4.2.Системи управління базами даних

Для збереження даних в різних мікросервісах буде використано різні БД в залежності від потреб і особливостей функціонування кожного окремого мікросервісу.

MS SQL Server – це системи управління реляційними базами даних, розроблена компанією Microsoft [16]. Містить в собі типову для таких продуктів функціональність для роботи з даним. Дана система управління БД випускається з комерційною ліцензією, тому при використанні її в цілях бізнесу необхідно використовувати платну ліцензією, безплатні версії доступні з обмеженою функціональністю. Компанією розроблено більше 15 випусків MS SQL Server для задоволення різноманітних потреб користувачів та з різною ціновою політикою. Така велика кількість випусків дозволяє системі працювати як з невеликими застосунками, створеними для персонального використання, так і з величезними застосунками, які кожен використовують мільйони користувачів.

Остання LTS версія MS SQL Server вийшла в 2017 році. В цій версії додано можливість використовувати систему в Linux середовищі та, що є особливо важливим в рамках даного проекту, в середовищі Docker як контейнер. У 2019 році випущено нову версію, однак офіційного випуску на момент написання документації не відбулося.

Для збереження даних в MS SQL Server використовується, як і в будь-якій реляційній БД, колекція таблиць з типізованими колонками. Система управління підтримує такі типи даних, як Integer, Float, Decimal, Char, Varchar, binary, Text та інші. Також користувач може самостійно створювати типи даних за необхідності. Вся статистика та логи транзакцій автоматично зберігаються та налаштовуються в рамках системи. При необхідності можна додавати процедури, індексатори, тригери та інші потрібні обєкти для управління даними в рамках MS SQL Server.

Структурно дані розділені на так звані “сторінки”, кожна з яких має розмір 8 Кб. Це найменша базова частина в структурі MS SQL Server, для якої доступні операції записування та зчитування даних. В кожній з таких сторінок є 96-байтний заголовок, який включає в себе метадані про “сторінку” – її тип, номер, вільне місце та ідентифікатор об'єкта, який є власником сторінки.

Управління вільним місцем в фізичному сховищі виконується за допомогою “екстентів” – структурних частин MS SQL Server, які містять в собі 8 “сторінок”. Об'єкт в базі даних може повністю займати цілий “екстент”, або ділити його з іншими об'єктами, але не більше 7, оскільки в одній “сторонці” зберігаються дані, які відносяться лише до одного об'єкта. Однак, об'єкти строк таблиць в MS SQL Server мають максимально допустимий розмір, і він рівний одній “сторінці” – 8 Кб. Виключенням є тільки строки та бінарні дані, які можуть зберігатися послідовно, розбиті на декілька “сторінок”, що належать до одного “екстенту”.

MS SQL Server використовує для запитів розроблену компанією Microsoft процедурну мову запитів – Transact-SQL (T-SQL). Ця мова є розширенням для стандартного набору інструкцій DDL та DML.

MS SQL Server має додаткові сервіси, що розширюють функціональність бази даних. Ці сервіси контролюються як будь-які інші Windows сервіси та мають API для взаємодії з ними. Приклади сервісів:

- сервіс машинного навчання;
- сервіс синхронізації;
- сервіс аналізу;
- сервіс повідомлень;
- інтеграційний сервіс;
- сервіс повнотекстового пошуку;
- SQL Server Management Studio.

Для використання даних, збережених в таблицях MS SQL Server, мікросервісом, створеним з використанням .NET Core, можна застосувати Entity Framework Core з драйвером для MS SQL Server.

PostgreSQL – система управління об’єктно-реляційною базою даних. Є найбільш розвиненою та функціональною базою даних з відкритим кодом [17]. Розробка PostgreSQL ведеться на базі Каліфорнійського Університету. В

перших версіях система була розрахована на запуск в UNIX-подібних системах. Але зараз існують версії для запуску PostgreSQL на операційних системах Windows, Mac OS X & Solaris. Вихідний код є у вільному доступі і розповсююджується під ліцензією PostgreSQL. Розробники можуть вільно використовувати, вносити зміни та розповсюджувати систему у будь якій формі. Завдяки стабільності системи та низькому відсотку посилок у вихідному коді PostgreSQL потребує мінімум затрат на підтримку та розгортання.

До переваг PostgreSQL над іншими системами управління реляційними сховищами даних можна віднести такі функціональні можливості:

- типи, створювані користувачем;
- наслідування таблиць;
- точки збереження в транзакціях;
- контроль багатоверсійності даних;
- асинхронні запити;
- вбудована підтримка Microsoft Windows Server;
- можливість бекапу системи при наявності точки збереження;
- перевагою системи також є можливість розширення системи власними функціями. Такі функції можна створювати на вбудованій мові PL/pgSQL, але вона має досить обмежені можливості. Тому розробники найчастіше додають функції, створені за допомогою «класичних» мов програмування, наприклад C, C++, C#, Java, Python та інші.

PostgreSQL створювалась загалом як розширювана система, тому розробники і користувачі системи мають широкі можливості, що дозволяють додавати власні типи даних, типи індексів, підтримку інших мов програмування для написання функцій та інше. Якщо певна частина системи сповільнює швидкодію або функціональність цієї частини вам потрібно

змінити в рамках розроблюваного проекту – можна розробити спеціальний плагін для покращення системи. Наприклад, в залежності від специфіки проекту можливе прискорення швидкодії при діях з базою за допомогою додавання нового оптимізатору.

При виникненні питань стосовно будь якої частини PostgreSQL можна звернутись до активної спільноти розробників системи. При необхідності негайної допомоги є можливість звернутись до однієї з багатьох комерційних організацій, що надають такі послуги.

Серед відомих компаній, що використовують PostgreSQL для своїх продуктів є Apple, Fujitsu, Red Hat, Cisco, Juniper Network, та інші.

MongoDB – це документо-орієнтована система керування базами даних. Має відкритий сирцевий код, розповсюджується за ліцензією AGPL. Відноситься до NoSQL баз даних, хоча формально займає нішу між реляційними базами даних та швидкими масштабованими система, які використовують схему ключ/значення для операцій даними.

Вихідний код системи написаний на мові C+.

Зберігання даних відбувається у JSON-подібному форматі. Всі файли мають бінарний формат BSON. Пошук, зберігання та інші взаємодії з файлами виконуються з використанням протоколу GridFS.

Redis – це сховище даних в оперативній пам’яті з відкритим вихідним кодом [18]. Найчастіше його використовують як базу даних, кеш або систему для передачі повідомлень. Підтримує такі типи даних як рядки, списки, хеші, набори, растрові зображення, відсортовані набори з діапазонними запитами та інші. Redis має вбудовану систему реплікації, скрипти Lua, транзакції та різні рівні збереження даних. Висока доступність та швидкість доступу до даних забезпечується Redis Sentinel та автоматичне розбиття з допомогою Redis Cluster.

Для кожного з типів існують атомарні операції, наприклад, додати символи до строки, інкрементувати хеш, додати елемент в список, розрахувати

перетин, об’єднання та різницю між наборами даних або вибрати перший елемент у відсортованому масиві.

Задля забезпечення найвищою швидкодії Redis використовує оперативну пам'ять для збереження даних, однак при необхідності дані можна також додатково зберігати на диску, а всі команди логувати. Додаткове збереження даних та логування можна вимкнути при необхідності використання Redis як звичайного кешу в оперативній пам’яті.

Redis має підтримку асинхронних операцій з швидкою неблокуючою першою синхронізацією та автоматичним підключенням при частковій розсинхронізації.

Redis можна використовувати з більшістю мов програмування. Написана системи на ANSI C і працює з більшістю POSIX систем, таких як Linux, *BSD, OS X. Для Windows офіційної підтримки немає, але Microsoft розробив та підтримає Win-64 версію Redis.

4.3.Брокер повідомлень

RabbitMQ – це кросплатформенний брокер повідомлень з відкритим вихідним кодом [19]. Розповсюджується за ліцензією Mozilla Public License. Останній випуск LTS версії відбувся у 2019 році. Написаний на мові програмування Erlang.

RabbitMQ має підтримку асинхронного обміну повідомленнями. Обмін проходить з використанням різних протоколів в залежності від потреб користувача.

Протоколи для обміну, які використовує поточна версія:

- AMQP 0-9-1;
- STOMP;
- MQTT;

- AMQP 1.0;
- HTTP;
- WebSockets.

Для зменшення навантаження на обробники повідомлень в RabbitMQ реалізована черга повідомлень, що дозволяє поетапно зчитувати їх і виконувати необхідні дії. Також, існує функціональність для паралельного зчитування повідомлень з черги та їх виконання. Особливо важливою ця перевага є у вебзастосунках, оскільки відповідь від сервера користувач повинен отримати якомога швидше, але деякі комплексні зміни можуть займати багато часу.

На відміну від

Для зручності використання при розробці створено драйвери для найбільш популярних мов програмування: Java, .NET, PHP, Python, JavaScript, Ruby, Go, та інші.

Додатково RabbitMQ надає API та інтерфейс для отримання інформації та зміни налаштувань. При необхідності в брокер повідомлень можна додати свій плагін для розширення функціональності або кращої інтеграції в існуючу систему. RabbitMQ підтримує розгортання як контейнер та може горизонтально масштабуватись при необхідності.

4.4.Docker хост

Docker – це платформа для управління ізольованими пакетами, які називаються контейнерами, на основі операційних систем Linux та Windows. Програмним забезпеченням для хосту контейнерів є Docker Engine [3]. Перша версія Docker була випущена в 2013, поточна LTS версія вийшла 11.04.2019. Сервіс має як платні, так і безплатні підписки на використання. Сирцевий код Docker написано на Go в поширюється під ліцензією Apache 2.0.

Контейнери є повністю ізольованими один від одного та обєднують в собі програмне забезпечення, сховища даних, бібліотеки та конфігураційні дані. Для взаємодії з іншими контейнерами використовуються визначені канали зв’язку.

Всі контейнери працюють на одному ядрі операційної системи, є більш легковаговими та мають кращу швидкодію у порівнянні з віртуальними машинами. На додаток до можливостей, що надаються ядром Linux (насамперед, cgroups і імен), контейнер Docker, на відміну від віртуальної машини, не вимагає або не включає окремої операційної системи. Натомість, він спирається на функціональність ядра і використовує ізоляцію ресурсів для процесора і пам'яті, і окремі простори імен, щоб ізолювати образ застосунку від операційної системи. Docker отримує доступ до функцій віртуалізації ядра Linux або безпосередньо за допомогою бібліотеки libcontainer, яка доступна з Docker 0.9, або опосередковано через libvirt, LXC (контейнери Linux) або systemd-nspawn.

Контейнери формуються з образів, в яких визначається вміст контейнеру. При роботі з образами поширеною є практика розширення уже існуючих образів, які можна завантажити з відкритих репозиторіїв, додатковими компонентами. Образ для контейнеру є шаблоном з усіма налаштуваннями, а, оскільки контейнери повністю відокремлені, з образу можна створити необхідну кількість однакових контейнерів. Це явище називається горизонтальним масштабуванням та дозволяє стабілізувати швидкодію системи перерозподіливши навантаження з одного контейнеру на велику кількість інших.

Для створення образу використовується Dockerfile, в якому зберігаються всі інструкції щодо його формування. В цьому файлі обовязково включають базовий шар образу, наприклад ubuntu або .net core sdk.

Docker Compose – це інструмент для визначення та запуску багатоканальних застосунків Docker. Він використовує файли YAML для

налаштування служб застосунків і виконує процес створення та запуску всіх контейнерів за допомогою однієї команди. Утиліта CLI-докера дозволяє виконувати команди для декількох контейнерах одночасно, наприклад, створювати образи, масштабувати контейнери, запускати зупинені контейнери та багато іншого. Команди, пов'язані з маніпуляцією образами або інтерактивними параметрами користувача, не мають відношення до Docker Compose, оскільки вони адресують лише один контейнер. Файл docker-compose.yml використовується для визначення служб застосунку та включає різні параметри конфігурації. Наприклад, параметр build визначає параметри конфігурації, такі як шлях до Dockerfile, параметр команди дозволяє перевизначити команди Docker за замовчуванням, тощо. Перша публічна версія Docker Compose (версія 0.0.1) була випущена 21.12.2013. Першу LTS версію (1.0), було випущено 16.10.2014.

Docker Swarm забезпечує функціональність кластеризації для контейнерів Docker, що перетворює групу Docker Engines в єдиний віртуальний Docker Engine. У Docker 1.12 і вище, режим Swarm інтегрований з Docker Engine. Утиліта Swarm CLI дозволяє користувачам запускати контейнери Swarm, створювати маркери виявлення, перераховувати вузли в кластері та багато іншого. Утиліта CLI Docker вузла дозволяє користувачам запускати різні команди для керування вузлами в Swarm, наприклад, перераховуючи вузли в Swarm, оновлюючи вузли і видаляючи вузли з Swarm. Docker керує вузлами, використовуючи алгоритм консенсусу Raft. Згідно з Raft, для того, щоб оновлення було виконано, більшість вузлів Swarm повинні погодитися на оновлення.

В рамках дипломного проекту ця платформа є виключно важливою при розробці серверної архітектури. Docker зараз нерозривно повязаний з поняттями “контейнеризація” та “мікросервіси”. Структурно в рамках Docker хосту кожен мікросервіс буде запускатись як окремий контейнер, а каналами зв’язку між ними будуть HTTP запити на API інших мікросервісів. Між собою

мікросервіси також будуть мати прямий звязок за допомогою брокеру повідомлень.

Висновки до розділу

Отже, проаналізувавши всі за та проти використання мікросервісної архітектури та детально опрацював завдання до розробки функціональності було обрано мікросервісну архітектуру через її надійність, масштабованість та новизну підходів та самої архітектури (мікросервісна архітектура у відомому нам вигляді з’явилась на початку 2010–х.).

5. РЕАЛІЗАЦІЯ ПРОЕКТУ

5.1.Створення архітектури проекту

5.1.1. Загальна схема компонентів

Загалом весь проект буде поділений на 2 умовні частини – клієнтська та серверна, запущена з використанням Docker. Схему компонентів, адаптовану для кращого розуміння можна побачити на рисунку 5.1. Також, схема компонентів візуалізовано у додатку А.

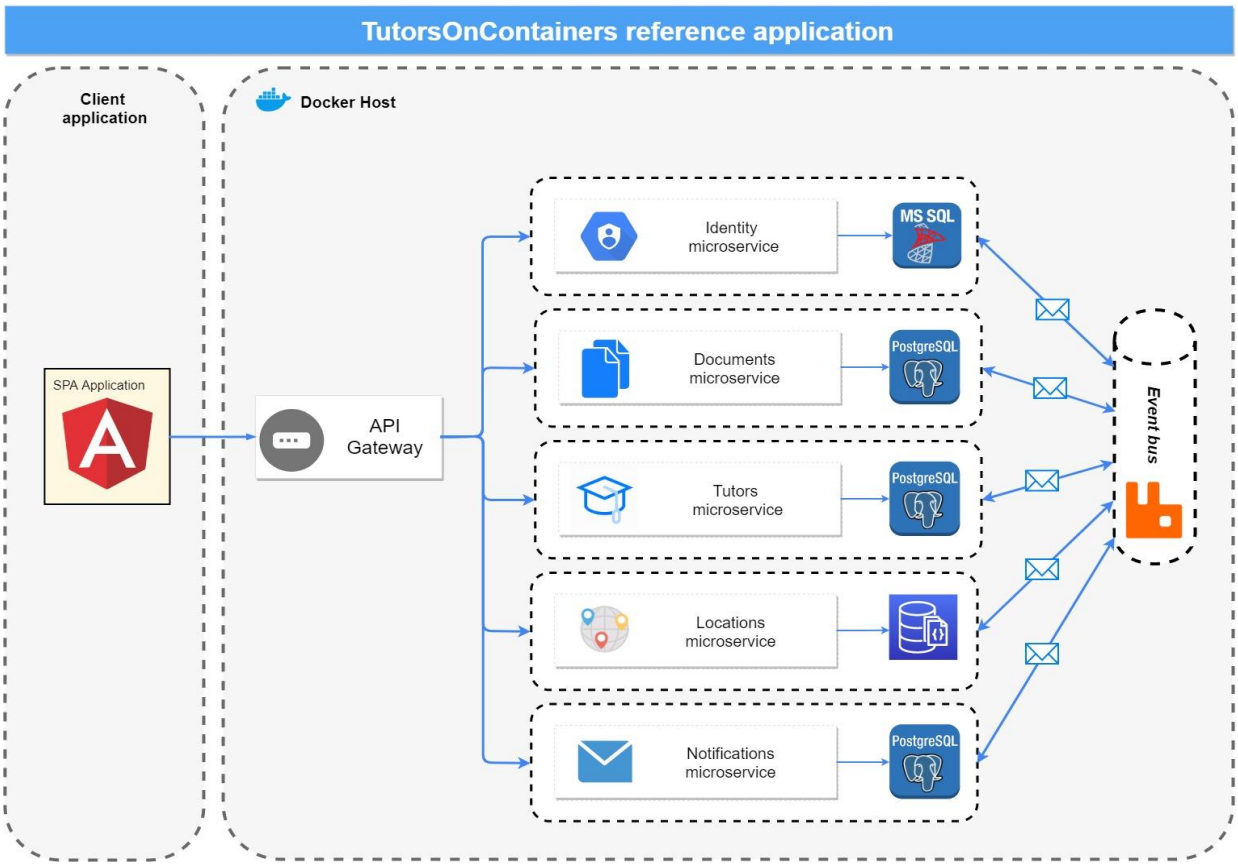

Рисунок 5.1 – Схема компонентів проекту

До клієнтьської частини будуть відностись будь які застосунки, які створені для користувача системи та мають інтерфейс. Наприклад, застосунки

для телефонів на базі Android та IOS або SPA-застосунки, розроблені з використанням фреймоврку Angular або бібліотеки React.

Для створення клієнтської частини в рамках реалізації проекту було розроблено SPA-застосунок на базі технології Angular версії 6.0.0. Як можна побачити з рисунку, зв'язок SPA-застосунок з серверною частиною проходить через один компонет, API Gateway, до функціональності якого входить забезпечення взаємодії компонентів клієнтьскої частини проекту з серверною частиною.

Застосунок відповідає за всі взаємодії з користувачем. Весь інтерфейс буде представлений саме в ньому. Також в функціональність застосунку, окрім обробки взаємодій з користувачами, входить початкова валідація даних, обробка повідомлень з сокетів, відображення коректних повідомлень про помилку при отриманні неправильних даних з серверної частини або даних, які містять в собі код серверної помилки.

При необхідності, до клієнтської частини можна легко додати новий застосунок, який може використовути готові API для взаємодії з серверною частиною.

Серверна частина містить в собі API Gateway, мікросервіси та шину подій. Опишемо окремо кожен з компонентів, його роль у загальній системі та функціональність, яку він буде виконувати.

5.1.2. Компонент первинної обробки запитів

API Gateway – це застосунок в рамках серверної частини, що містить в собі API для взаємодії всіх компонентів клієнтської частини з серверною та побудований на основі однойменного патерну. Коли користувач робить запит до серверної частин, то перш за все цей запит потрапляє до цього компоненту. До функціональності API Gateway входить обробка всіх запитів та виконання сценаріїв взаємодії та використання даних з мікросервісів. Клієнт за

допомогою компонетів клієнтської частини не може напряму звертатись до мікросервісів, всі взаємодії обробляються за допомогою компоненту API Gateway.

Варто зауважити що даний компонент не входить до мікросервісів, а є тільки проміжним елементом між мікросервісами та клієнтськими компонентами системи. В API Gateway немає власної бази даних, в ньому не зберігається нічого, його завданням є переадресація запитів користувачів на відповідні мікросервіси.

Компонент розроблений як .NET Core WEB проект з використанням шаблону WEB API. Доступ до компоненту з клієнтсько частини проходить за допомогою протоколу HTTP. При розширенні системи та її функціональності даний компонент можна розбити на 3 частини, згідно з патерном BFF (Backends for Frontends). Приклад використання даного патерну можна побачити на рисунку 5.2.

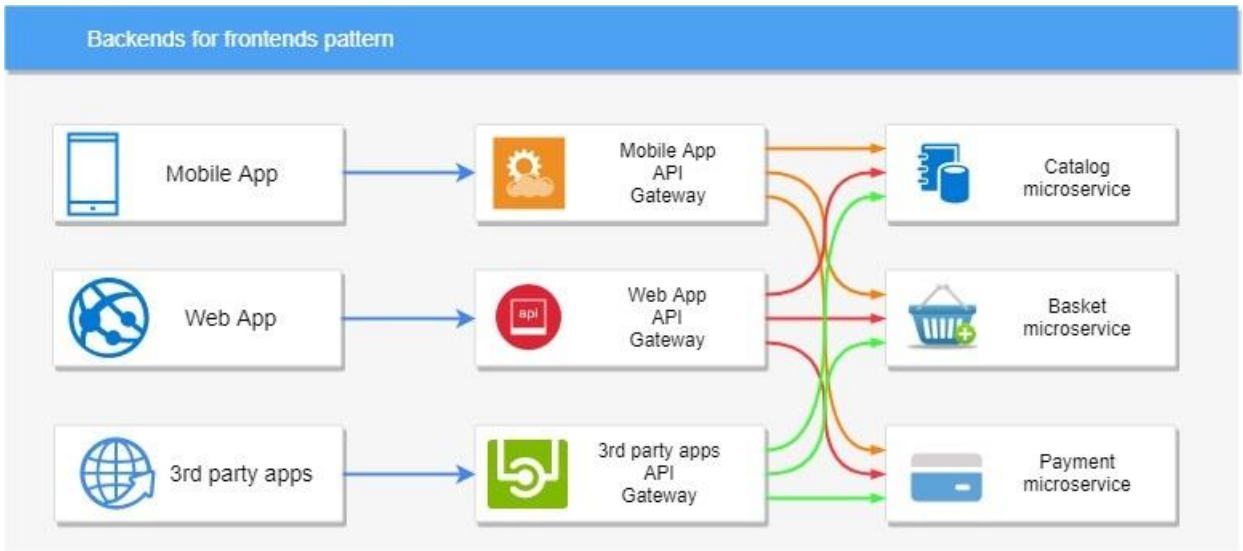

Рисунок 5.2 – Приклад використання патерну BFF

В рамках реалізації даного патерну створюються 3 компоненти, кожен з яких відповідає за свою частину доступу до даних системи. Компонент WEB App API Gateway обробляє запити від сайтів та відповідно користувачів, які

використовують систему через браузер. Mobile App API Gateway використовується для доступу до даних через мобільні застосунки системи. Public API Gateway може бути використано третіми сторонами для доступу до даних системи. Цей патерн виправдовує своє використання при наявності багатьох видів застосунків дял користувачів і в рамках розробки проекту не потрібен.

5.1.3. Компонент синхронізації даних

Event bus – це шина даних, яка розроблена для синхронізації даних. У мікросервісах збереження даних виконується децентралізовано, у кожному є своя база даних. Однак система залишається єдиним цілим, тому дані, збережені у різних сервісах є звязаними між собою. При такій побудові системи збереження даних можуть виникнути випадки, коли необхідно синхронізувати записи в базах, оскільки зміни в одному мікросервісі впливають на дані, збережені в іншому. Для зміни даних в іншому сервісі можна використати сценарії, які виконує компонент API Gateway, але це призведе до ускладнення розуміння зв’язків між сервісами та збільшенню кількості запитів, що негативно вплине на швидкодію.

Тому для вирішення проблеми синхронізації даних між сервісами створено додатковий компонент серверної частини проекту. Синхронізація даних відбувається за допомогою технології Rabbit MQ. Патерн subscriber-receiver дозволяє сервісами «підписуватись» на отримання повідомлень від інших сервісів. Для доступу до даного компоненту мікросервіси використовують протокол, створений на основі TCP (AMQP/STOMP). Створення для цих цілей окремого компоненту дозволяє не впливати на швидкодію та синхронізувати дані в автоматичному режимі.

Прикладом може послужити синхронізація даних між мікросервісом Identity, в якому реалізовано функціонал по збереженню основних даних всіх

зареєстрованих користувачів, та мікросервісом Notifications, в якому реалізовано функціонал відправки повідомлень користувачам на телефони, електронну пошту, інше. В мікросервісі Identity зберігається телефон користувача. Однак при спробі надіслати повідомлення за даним номером мікросервіс Notifications повернув помилку, яка свідчить про те, що такого номера телефону не існує. Після обробки відповіді від стороннього сервісу для надсилання повідомленнь мікросервіс Notifications формує повідомлення для Identity з вказівкою позначити даний номер як некоректний. Дане повідомлення надсилається до Event bus в канал повідомлень для мікросервісу Identity. Оскільки Identity є підписником на повідомлення з даного каналу, він отримує інформацію і, обробивши її, змінює дані в базі. Обробка такої помилки в API Gateway призведе до ще одного запиту до Identity мікросервісу, що сповільнить отримання даних клієнтською частиною.

5.1.4. Компоненти мікросервісів

В системі представлено 5 мікросервісі, які розділяють між собою функції, які були описані в вимогах до розроблюваної системи. Всі мікросервіси запускаються за допомогою Docker хосту як окремі контейнери.

Для виділення кожного з мікросервісів було використано методи, описані в розділі. При розширенні системи та необхідності розбиття на менші частини поточні компоненти серверної частини проекту, які є мікросервісами, можуть з легкістю бути розділені на компоненти з меншою функціональністю.

Identity мікросервіс – компонент системи, в якому зберігаються всі основні дані про користувачів. Саме з використанням даних цього мікросервісу користувач може авторизуватись або зареєструватись в системі. Через це стабільність даного компоненту є критично важливою для функціонвання системи, тому при його розгортанні в на сервері необхідно мінімізувати можливість перебоїв в роботі. В базі даних даного сервісу зберігаються

електронні пошти, імена, телефони, секретні коди для двофакторної авторизації. Також даний сервіс відповідає за передачу даних до API Gateway, з яких буде сформовано JWT-токен.

Компонент розроблений як .NET Core WEB проект з використанням шаблону WEB API. Доступ до компоненту з API Gateway проходить за допомогою протоколу HTTP. Для збереження даних використовується система управління реляційними сховищами даних MS SQL.

Document мікросервіс – це компонент, що відповідає за зберігання будь яких даних на сервері. В функціональності даного мікросервісу реалізовано можливість збереження навчальних матеріалів на сервері, додавання документів для підтвердження особи користувача системи та документів, що доводять вину іншого користувача при виникненні суперечки. При розгортанні даного компоненту на сервері необхідно звернути особливу увагу на необхідність додаткового місця в пам’яті серверу, оскільки всі документи будуть напряму зберігатися на сервері.

Також даний сервіс є небезпечний тим, що як документ можна завантажити файл, в якому будуть міститись команди, що можуть нашкодити системі або допомогти зловмиснику викрасти важливі дані при спробі відкрити або надіслати користувачу файл. Через це в майботньому потрібно додати можливість компоненту перевіряти вміст файлу на потенційно небезпечні комнади.

Компонент розроблений як .NET Core WEB проект з використанням шаблону WEB API. Доступ до компоненту з API Gateway проходить за допомогою протоколу HTTP. Для управління документами всі дані про внесені зміни та дії з ними зберігаються у сховищі даних під управлінням PostgreSQL.

Tutors мікросервіс – це компонент, в якому відбувається обробка та збереження основних даних, які відносяться безпосередньо до навчального процесу. Дані про репетиторів, їхня спеціалізація, відгуки та час, коли

репетитор готовий прийняти учня. Також загальні дані для проекту зберігаються в цьому компоненті як словники, наприклад, список предметів та спеціалізацій.

Даний компонент є найбільш об’ємним за кількістю реалізованих в ньому функцій. Через це Tutors мікросервіс при необхідності буде найпростіше розбити на менші сервіси. Тому при розробці мікросервісу використовувався принципом модульності, що полегшить подальшу розробку та виділення з цього мікросервісу інших сервісів.

Стабільність даного компоненту також критично важливо для коректної роботи системи, тому при розгортанні його на сервері потрібно забезпечити високу стабільність.

Компонент розроблений як .NET Core WEB проект з використанням шаблону WEB API. Доступ до компоненту з API Gateway проходить за допомогою протоколу HTTP. Для збереження даних використовується система управління реляційними сховищами даних PostgreSQL.

Locations мікросервіс – це компонент системи, що відповідає за використання сторонніх сервісів для визначення положення юзера та синхронізацію розбіжностей у часі між різними користувачами. Хоча графік кожного окремого репетитора зберігається в міркосервісі Tutors, саме Locations відповідає за коректне відображення часу у кожного з користувачів незалежно від часового поясу.

Даний мікросервіс використовує сторонні API для отримання необхідних даних. Кожен з сервісів надсилає дані у різному вигляді, тому для їх збереження використано документну нереляційну базу даних Mongo DB.

Даний компонент не є надзвичайно важливим і стабільність його роботи не має критичного впливу на систему. При виникненн проблем в Locations мікросервісі його дані можуть бути тимчасово замінені в API Gateway компоненті даними з Tutors мікросервісу.

Однак даний компонент є критичним місцем в швидкодії, адже використання сторонніх сервісів не гарантує швидкість відповіді від них. Тому при запитах до цього компоненту використано асинхронність та можливість пропустити запит і продовжити роботу по обробці запиту користувача якщо затримка в отриманні даних від стороннього сервісу занадто велика.

омпонент розроблений як .NET Core WEB проект з використанням шаблону WEB API. Доступ до компоненту з API Gateway проходить за допомогою протоколу HTTP.

Notifications мікросервіс – це компонент, що відповідає за надсилання користувачам системи повідомлень. Як і попередній мікросервіс, використовує різні сторонні сервіси для надсилання. В реалізації створено можливість відправки повідомлень до користувачів за допомогою електронної пошти та СМС-повідомлень. В майбутньому можливо розширити даний компонент з використанням додаткових сервісів для інформування користувачів. В базі даних сервісу збережені налаштування кожного з користувачів відносно сповіщень та підписки на новини та оновлення системи.

Компонент не є критично важливим для системи, оскільки при збоях в його роботі проблеми виникнуть тільки при спробі надсилання повідомлень користувачу.

Компонент розроблений як .NET Core WEB проект з використанням шаблону WEB API. Доступ до компоненту з API Gateway проходить за допомогою протоколу HTTP. Для збереження даних використовується система управління реляційними сховищами даних PostgreSQL.

5.2.Проектування системи

При розробці робочого прототипу проекту для кожного мікросервісу, окрім мікросервісу Locations, було створено схему бази даних. Приклад схеми бази даних для мікросервісу Tutors можна переглянути у додатку Г.

Враховуючи особливості функціонування мікросервісної архітектури, деякі поля, наприклад TutorID, що на перший погляд мали би бути зовнішніми ключами, не посилаються на таблиці, оскільки дані збережені в іншому сервісі. Зв'язування даних відбувається в компоненті превинної обробки даних (API Gateway), а синхронізація – через брокер повідомлень (Service bus).

Для розробки бази даних було використано підхід Code-first. Цей підхід заключається в тому, що для створення таблиць і загалом бази даних використовуються моделі, які написані на мові C#. Перевагою цього підходу є використання для розробки тільки C#. Моделі для отримання даних в будь-якому випадку потрібно буде створювати, але при використанні іншого підходу доведеться робити одну і ту саму роботу двічі: створення моделей та створення таблиць в базі даних за допомогою.

Хоча цей підхід має і певні недоліки: при такому підході швидкість роботи з базою даних дещо менша, оскільки запити до сховища даних виконуються з використання LINQ to Database. LINQ (Language-Integrated Query) – це частина мови програмування C#, яка додає можливості повнофункціонального та зручного підходу в створенні запитів до будь яких колекції даних. Також LINQ допомагає полегшити обробку даних та обробляти дані. В певній мірі це схоже на розширену версію SQL, інтегровану в C#.

Запити, написані на LINQ to Database для отримання даних з бази транслюються в SQL. Цей процес залежить від драйверу, який використовується в фреймоворку, що забезпечує доступ до даних. Драйвер не підтримує всі сценарії, що робить SQL в певних випадках більш функціональним. До того ж, часто при написанні запитів на LINQ драйвер транслює його не найоптимальнішим шляхом, що зменшує швидкодію у порівнянні з скриптами, написаними вручну.

Для використання LINQ потрібно перш за все зв'язати дані моделі з таблицями в базі даних. Не дивлячись на те, що таблиці створюються з моделей, для їх підключення потрібно додати їх до файлу налаштувань. Для

створення будь яких налаштувань потрібно підключити базу даних до проекту. Додавання будь яких сервісів та модулів виконується в Startup файлі, що використовується для створення веб-хосту в файлі Program. Така структура є стандартно для проектів, розроблених з використанням .NET Core версії 2 і вище. В Startup файлі виконується підключення всіх middleware, websocket та обробників запитів. При небхідності додається авторизація та аутентифікація користувача. Якщо веб-застосунок потребує використання бази даних або стороннього модуля, він має бути доданим до колекції сервісів, що ініціалізується перед початковим запуском хосту.

Для підключення бази даних до Startup клас потрібно використати метод AddDbContext. Цей метод застосовується для IServiceCollection і додає сховище даних. AddDbContext є generic методом, де типом, що передається в метод, є клас, який наслідує базовий клас для контексту бази даних DbContext.

В даному файлі ініціалізуються всі налаштування для підключення бази даних. Кожна з таблиць підключається як DbSet. Приклад підключення таблиці Reviews:

public DbSet<Review> Reviews { get; set; }

Дана властивість класу є generic-властивістю, в яку передається клас моделі. По стандартним налаштуванням Entity Framework всі властивості, що є класі автомAPIнгом додаються до SQL запиту, що створює таблицю. При необхідності особливих налаштувань для мапінгу поля використовується анотація.

Деякі можливості анотації при створенні класів моделей [20]:

- ключове поле: [Key];
- поле з забороненим null-значеням: [Required];
- поле, яке не буде додано в модель: [NotMapped];
- поле, що позначає зовнішній ключ для об’єкту, і використовується для зв’язування даних: [ForeignKey("LessonId")];

- поле для синхронізації даних: [Timestamp];
- уточнення особливостей створення поля в таблиці бази даних: [Column(TypeName = "decimal(20, 10)")]. В даному випадку кількість знаків при створенні колонки в таблиці бази даних буде обмежена.

Можливо використовувати не тільки готові атрибути, а і створювати свої. Наприклад, існують поля в моделі, які, при створенні таблиці повинні бути збережені в окремій колонці, однак користувач ні в якому разі не повинен отримати доступ до цих даних. При запитах об'єкти, отримані з бази даних одразу перетворюються в моделі. Щоб полегшити роботу та не фільтрувати кожен окремий запит можна створити свою анотацію для фільтрування даних на рівні властивостей моделі. Після цього необхідно додати пост-обробник даних, що передаються користувачу, і виключати поле, яке позначено даною анотацією з даних, що надсилаються. Прикладом такого поля може слугувати токен для підключення користувачу Google-аутентифікації. Часто цей вид аутентифікації використовують як додатковий захист на самих важливих частинах систем. Користувачу у будь-якому випадку цей токен не може бути потрібен, єдина його ціль – забезпечити поточній системі можливість проводити аутентифікацію користувача з використання Google. При потраплянні цього токену до рук зловмисників виникає загроза безпеці даних користувача. Тому необхідно захистити це поле.

Саме для таких цілей створено анотацію [ExcludeFromResponse]. Ця анотація позначає поле як «Виключене з відповіді». Щоб анотація працювала необхідно додати обробник. Цей клас повинен бути наслідником DefaultContractResolver та перевизначати метод CreateProperties. В цьому методі і проходить обробка моделі перед тим, як вона буде серіалізована в Json формат та відправлена користувачу. За допомогою LINQ налаштуємо фільтр властивостей моделі:

```
var includedPropList = type.GetProperties()
    .Where(prop => !Attribute.IsDefined(prop, typeof(ExcludeFromResponse)))
```

```
    .Select(x => x.Name)
    .ToList();
```

Таким чином створюється модель, з якої у подальшому буде створена таблиця в базі даних і через яку буде проходити зв’язування даних.

Підключивши всі необхідні моделі і передавши клас, що наслідує DbContext до методу AddDbContext як generic-тип, необхідно передати строку підключення до бази даних як параметр цього методу. Найчастіше дані про підключення зберігаються в окремому конфігураційному файлі. Це робиться з ціллю можливості легкої зміни користувача або паролю до базі без потреби перекомпіляції вихідних файлів, оскільки файли налаштувань, разом з скомпільованими бібліотеками застосунку, потрапляють до вихідної папки.

Типова строка підключення виклядає наступним чином:

"User ID=postgres; Password=1234; Host=127.0.0.1; Port=5432; Database=tutorsMain; Pooling=true;"

Дана строка дозволяє підключитись до бази даних PostgreSQL за адресою 127.0.0.1, що є машиною користувача, на порті 5432, який є стандартним для цієї бази даних, з використанням вхідних даних користувача та його пароля. Також уточнюється ім’я бази даних та додаткові параметри.

Окрім строки підключення в параметрах методу можна передати рівень логування помилок, що дозволить при розробці аналізувати ряд помилок, які виникають на стороні бази даних.

Також важливим параметром є визначення життєвого циклу сервісу. В рамках фремворку .NET Core існують 3 можливих життєвих цикли:

- Transient – цей вид життєвого циклу створює об’єкт класу при кожному запиті до сервісу на його створення. Це означає, що в ході одного запиту сервіс буде створено стільки разів, скільки до нього звернуться різні компоненти системи;
- Scoped – це вид життєвого циклу, при якому об’єкт сервісу створюється при першому виклику під час запиту і в подальшому використовується

саме цей об’єкт. При завершенні циклу запиту об’єкт знищується з допомогою garbage collector;

- Singleton – обєкт створюється при першому запиту, в якому він необхідний і існує в рамках застосунку поки він безпереривно працює.

Для баз даних використовується Transient, що дозволяє створювати кожен раз нове підключення при необхідності. Навіть у рамках одного запиту. Це дозволяє прискорити роботу та стійкість системи, оскільки навіть критична помилка в одному з запитів до бази даних не зупинить роботу всіх інших і дозволить отримати їм дані.

Окремо варто відмітити підключення Redis до мікросервісів застосунку. На відміну від інших баз даних, бібліотека для Redis містить в собі спеціальний метод AddDistributedRedisCache, в який передаються самі базові конфігураційні дані для успішного підключення Redis до проекту.

Підключивши всі необхідні бази та перевіривши налаштування можна стоврювати таблиці з моделей. Для цього використовується Entity Framework та поняття «міграції» в рамках даного фреймворку. При першому запуску EF генерує файли, в яких всі моделі перетворює у набір викликів власних функцій. Всі параметри, створені автогенератором можливо змінити, однак завжди існує ризик отримати помилку через розсинхронізацію моделей даних та реально існуючих таблиць.

Для додвання міграцій використовують команду «dotnet ef migrations add». При необхідності автоматично буде створена папка migrations, в якій будуть зберігатись всі необхідні файли. Після додавання кожної міграції є можливість змінити її налаштування, знайшовши їх у папці. Файли матимуть назву міграції, яку розробник самостійно вказує при створенні. Якщо все створено вірно і можна вносити зміни до бази потрібно виконати команду «dotnet ef database update». Це внесе зміни до бази та змінить структуру.

Для використання LINQ потрібно визвати клас контексту, викликати властивість таблиці, що пов’язує через generic DbSet моделі в C# та таблиці в базі даних, та написати до неї запит.

В рамках проекту, для полегшення користування базою даних та використання готових функцій було створено GenericRepository<TEntity>. Цей клас має базові методи для роботи з певною таблицею та звязаними даними в базі. Це дозволить перемістити підключення клас контексту доступу до даних в універсальний репозиторій, що дозволить розробнику концентруватись на роботі одразу з певною моделлю даних, не перебираючи всі властивості в пошуках потрібної таблиці. Для отримання доступу до даних певної моделі потрібно просто додати створення специфічного репозиторію до конструктора класу. В Startup класі інверсія контролю зв’язує generic репозиторій з відповідною властивістю з класу контексту даних.

Методи в класі GenericRepository є універсальними та можуть використовуватись для роботи з будь якою моделлю. Створено методи для обробки даних напряму та через транзакції. Відповідні методи для роботи з використанням транзакцій та збереженням цілісності даних мають у своїй назві слово Transaction. Приклад коду оновлення моделі з використанням транзакцій та відкату змін при помилці:

```
public async Task UpdateTransaction(IEnumerable<TEntity> items) {
   var strategy = _context.Database.CreateExecutionStrategy();
   await strategy.ExecuteAsync(async () =>
   {
   using (var transaction = await _context.Database.BeginTransactionAsync()) {
      try {
         var entities = items as TEntity[] ?? items.ToArray();
         var data = entities.ToDictionary(p => p.Id);
         var baseline = _dbSet.Where(p => data.Keys.Contains(p.Id));
         var collection = typeof(TEntity).GetProperties();
```

```
            foreach (var entity in baseline) {
                var item = data[entity.Id];
                foreach (var field in collection) {
                    var inter = field.PropertyType.GetInterfaces();
                    var val = typeof(TEntity).GetProperty(field.Name).GetValue(item);
                    typeof(TEntity).GetProperty(field.Name).SetValue(entity, val);
                }
            }
            await _context.SaveChangesAsync();
            transaction.Commit();
        }
        catch (Exception ex) {
        transaction.Rollback();
    }
}
});
}
```

Цей код є універсальним та безпечним методом для оновлення даних, яких поверне старі дані при помилці, що збереже цілісність.

Повернемось до бази даних Tutors мікросервісу та детально розглянемо кожну таблицю представлену в ньому.

При розробці даного мікросервісу було створено наступні таблиці:

- Majors;
- Directions;
- Tutor_majors;
- Prices;
- Prices_hist;
- Lessons;

- Lessons_hist;
- Schedules;
- Schedules_hist;
- Reviews_hist;
- Reviews.

Також в мікросервісі додано ще декілька таблиць, але вони не містять зв’язків з іншими даними. Ці таблиці є «словниками», в яких зберігаються статичні дані, а також деякі інші дані, що відносяться до навчального процесу. З подальшого розгляду ці таблиці буде виключено, оскільки вони не представляють інтересу для розгляду.

В рамках функціонування системи всі предмети які можуть викладати репетитори розділяються по напрямкам та дисциплінам. Така схема розроблена для зручного використання при пошуку репетитора і більш точної спеціалізації, що є необхідним для викладання учням, які потребують вузькопрофільних знань з одного предмету.

Таблиця 5.1 - Структура таблиці «Directions»

Назва	Тип даних	ПК	ЗК	Опис поля
Id	number	+	-	Ідентифікатор
Name	varchar	-	-	Назва направлення
Code	varchar	-	-	Код направлення

Таблиця Directions містить в собі всі напрямки, з яких доступна навчання на платформі. Зміни до цієї таблиці можуть вносити тільки адміністратори системи, користувачі тільки бачать дані, які збережені в ній.

Таблиця 5.2 - Структура таблиці «Majors»

Назва	Тип даних	ПК	ЗК	Опис поля
Id	number	+	-	Ідентифікатор
DirectionID	number		+	Зовнішнє посилання на напрямок, до якого відноситься дана спеціалізація
Name	varchar	-	-	Назва направлення
Code	varchar	-	-	Код направлення

Таблиця, що містить в собі всі доступні для вибору репетиторів спеціалізації. При створенні профілю спеціаліст з індивідуального навчання вказує саме спеціальність, просто напрямок неможливо вказати. Однак всі спеціальності в обов'язковому порядку відносяться до певних напрямків. Користувач має змогу шукати за напрямком, оскільки система може містити не всі спеціальності. Отримавши список репетиторів, які працюють за обраним напрямком, користувач може робити запити до них з певними уточненнями стосовно своїх питань. Зміни до цієї таблиці можуть вносити тільки адміністратори системи, користувачі тільки бачать дані, які збережені в ній.

Таблиця 5.3 - Структура таблиці «Tutor_majors»

Назва	Тип даних	ПК	ЗК	Опис поля
Id	number	-	-	Ідентифікатор
TutorID	number	+	-	Ідентифікатор користувача, який пов'язує дану таблицю з даними в мікросервісі Identity
MajorID	number	+	-	Посилання на спеціальність

Таблиця, в якій зберігаються всі спеціальності репетиторів. При пошуку репетитора з критерієм спеціальності використовуються дані, збережені тут.

Через ідентифікатор користувача пов’язана з даними в іншому мікросервісі ,а саме Identity.

Таблиця 5.4 - Структура таблиці «Prices»

Назва	Тип даних	ПК	ЗК	Опис поля
Id	number	+	-	Ідентифікатор
TutorMajorID	number		+	Посилання на спеціальність репетитора
Price	decimal	-	-	Ціна за годину заняття
Additions	varchar	-	-	Додаткова інформація, текст, який вводить репетитор
VersionID	number	-	-	Поточна версія запису в таблиці
CreationDate	date	-	-	Дата створення
UpdateDate	date	-	-	Дата змінення

Prices – це таблиця, в якій зберігаються всі дані репетиторів про ціну за годину кожного з них в залежності від обраної користувачем спеціальності для вивчення. Також існує можливість вводу додаткових даних, наприклад, про акцію або знижку. Додаткові дані є текстовим полем, туди можна вводити будь-які дані, вони будуть відображені в інтерфейсі у відповідному полі.

Таблиця 5.4 - Структура таблиці «Prices_hist»

Назва	Тип даних	ПК	ЗК	Опис поля
Id	number	+	-	Ідентифікатор
PriceID	number		+	Посилання на спеціальність репетитора
OldPrice	decimal	-	-	Неактуальна ціна за годину заняття

Продовження таблиці 5.4

Назва	Тип даних	ПК	ЗК	Опис поля
OldAdditions	varchar	-	-	Неактуальна додаткова інформація
VersionID	number	-	-	Поточна версія запису в таблиці
CreationDate	date	-	-	Дата створення
UpdateDate	date	-	-	Дата змінення

Prices_hist – це таблиця, яка створена для моніторингу змін в даних та збереження історії всіх змін. Вона потрібна для вирішення можливих проблем та суперечок між користувачами через фінансові або якісь інші проблеми. Загалом в базі даних мікросервісу Tutors для всіх основних таблиць є додаткові таблиці. Всі вони в назві містять «_hist» і дублюють поля основної таблиці з посиланням на актуальний запис. Надалі описуватись ці таблиці не будуть, адже вони мають точно такий самий функціонал і цілі, як і Prices_hist.

Таблиця 5.5 - Структура таблиці «Schedules»

Назва	Тип даних	ПК	ЗК	Опис поля
Id	number	+	-	Ідентифікатор
Day	date		+	Дата, яка відповідає дню, для якого створено графік
Week	number	-	-	Порядковий номер неділі, потрібне для полегшеного сортування
StartTime1	date	-	-	Початковий час першої частини, коли репетитор стає вільним і може прийняти учнів

Продовження таблиці 5.5

Назва	Тип даних	ПК	ЗК	Опис поля
Duration1	Float	-	-	Тривалість першої частини доступного часу
StartTime2	date	-	-	Початковий час другої частини, коли репетитор стає вільним і може прийняти учнів
Duration2	float	-	-	Тривалість другої частини доступного часу
VersionID	number	-	-	Поточна версія запису в таблиці
CreationDate	date	-	-	Дата створення
UpdateDate	date	-	-	Дата змінення

Schedules – таблиця з графіками доступного для роботи часу репетиторів. На кожен день є свій запис, що дозволяє робити унікальний графік на кожен день і не бути залежним від понедільного графіку. Це додає гнучкості в роботі, до того ж, у більшості аналогів таке тонке налаштування неможливе.

Два періоди часу є найбільшим оптимальним періодом з точки зору швидкодії та розділення даних в базі. При необхідності змін користувачі завжди можуть домовитись про все у повідомленнях на базі платформі або у приватній розмові.

Таблиця 5.6 - Структура таблиці «Reviews»

Назва	Тип даних	ПК	ЗК	Опис поля
Id	number	+	-	Ідентифікатор
TutorID	number	-	+	Посилання на репетитора
LessonID	number	-	+	Посилання на певний урок
Rating	number	-	-	Рейтинг оцінки (від 1 до 5)

Продовження таблиці 5.6

Назва	Тип даних	ПК	ЗК	Опис поля
Comment	varchar	-	-	Коментар щодо відгуку прозаняття або репетитора
VersionID	number	-	-	Поточна версія запису в таблиці
CreationDate	date	-	-	Дата створення
UpdateDate	date	-	-	Дата змінення

Reviews – це таблиця, де зберігаються всі відгуки про роботу репетиторів. Користувач може залишити відгук як і загалом про репетитора, так і про конкретне заняття. Також репетитор може залишити свій відгук про учня.

Для даної таблиці створена таблиця моніторингу за змінами.

Таблиця 5.7 - Структура таблиці «Lessons»

Назва	Тип даних	ПК	ЗК	Опис поля
Id	number	+	-	Ідентифікатор
StartTime	date	-	-	Час початку заняття
Duration	float	-	-	Тривалість заняття
TutorID	number	-	+	Посилання на репетитора
PupilID	number	-	+	Посилання на учня
ScheduleID	number	-	+	Зовнішній ключ, який пов’язує кожен урок з певним днем графіку
TypeFlag	number	-	-	Тип уроку
MajorID	number	-	+	Посилання на спеціалізацію, за якою буде проводитись заняття
VersionID	number	-	-	Поточна версія запису в таблиці
CreationDate	date	-	-	Дата створення
UpdateDate	date	-	-	Дата змінення

Lessons – таблиця, в якій містяться записи всіх уроків, які є в системі. Саме до уроків прив’язуються навчальні документи та оцінки. Зберігає час початку та тривалість заняття. TypeFlag зберігає в собі статус поточного уроку. За допомогою цієї змінної користувач може бачити чи підтвердив репетитор його заявку на урок, якого типу буде урок – з використанням Skype чи особиста зустріч та інше.

Для даної таблиці створена таблиця моніторингу за змінами.

5.3.Взаємодія компонентів

Для пояснення на практиці як саме взаємодіють компоненти наведемо приклад відправлення повідомлення користувачем застосунку. В додатку Б представлена діаграма послідовності для цього випадку.

Отже, початкова послідовність дій системи під час запиту на надсилання повідомлення іншому користувачу:

- користувач робить HTTP POST запит до серверної частини, де в тілі запиту є JSON-об'єкт за повідомленням;
- запит потрапляє до API Gateway компоненту, де одразу ж перевіряється токен із заголовку;
- якщо токен не валідний – користувач отримує відповідне повідомлення, якщо все вірно – процес обробки буде продовжено.

Після перевірки токену користувача та отримання даних починається етап взаємодії з мікросервісами. Компонент API Gateway використовується як центральний елемент, в якому збережено всі сценарії та послідовності взаємодії з мікросервісами. На кожному етапі компонент перевіряє дані, отримані від мікросервісів та надсилає повідомлення користувачам і разі помилки. Розглянувши наступні етапи в діаграмі послідовності, отримуємо:

- наступним йде запит до Tutors мікросервісу для з’ясування взаємозв’язків користувачів. Оскільки дана ситуація розглядає варіант надсилання адресату повідомлення на телефон, користувач повинен мати активні взаємодії з адресатом в найближчий місяць;
- API Gateway отримує відповідь від Tutors та обробляє її. Якщо користувач не може надіслати повідомлення – він отримає відповідне повідомлення про неможливість здійснення операції. Якщо користувач має відповідні права, виконання запиту продовжується.

Варто зауважити, що сценарії виконання запитів до мікросервісів можуть мати послідовну або паралельну схеми виконання. Послідовна схема використана в даному випадку. Цей вибір зумовлений пов’язаністю даних, що отримуються з кожного мікросервісу. Якщо дані не пов’язані між собою, запити до мікросервісів можна виконувати асинхронно, що прискорює швидкодію та дозволяє обробляти дані, отримані від мікросервісів, по мірі їх отримання.

- на цьому етапі задіяний Notifications мікросервіс, який відповідає за відправку всіх повідомлень у системі. Об’єкт повідомлення користувача передається у цей мікросервіс;
- наступним кроком повідомлення зберігається в базі даних;
- після збереження повідомлення в базі Notifications компонент викликає сторонній сервіс для відправки повідомлення (Twilio);
- отриману відповідь сервіс повертає на API Gateway, який перевіряє чи успішно всі відбулось, і якщо помилка – повідомляє про це користувачу відповідним спливаючим вікном.

Таким чином можна зробити висновок, що основний обробник даних – API Gateway. Саме цей компонент робить запити до інших компонентів системи і веде взаємодію з клієнтським застосунком.

5.4. Тестування роботи програми

Оскільки інтерфейс системи готовий не повністю, для повноцінної перевірки та тестування роботи функціоналу можна використати Postman або аналогічний програмний засіб, що дозволяє створювати HTTP запити та виконувати їх.

Отже, для перевірки роботи програми необхідно:

- скачати Git репозиторій з проектом. Це можна зробити клонувавши проект з github-репозиторію[1] (після підтвердження доступу від власника сирцевого коду), або отримавши копію проекту іншим чином (наприклад, з електронного носія);
- встановити Docker (за необхідністю) з офіційного сайту[3] або з будь якого іншого ресурсу за вибором;
- збілдити всі компоненти програми згідно з інструкцією, яка знаходиться в кореневій папці репозиторію[1];
- запустити систему з допомогою Docker (використовуючи інструкцію);
- встановити Postman та увійти до системи авторизувавшись або створивши новий профіль користувача;
- перейти за посиланням[2] та додати документацію до середовища. Детальні інструкції щодо використання можна знайти на офіційному сайті Postman[4];
- обрати Environment – «tutorsOnContainers»;
- тестувати систему, використовуючи готові запити або запустити Angular app та тестувати функції, для яких додано інтерфейс користувача.

Варто зауважити що бази даних будуть пустими, тому потрібно буде власноруч їх заповнити.

Оскільки використовуються Docker-контейнери розробнику не потрібно самостійно встановлювати бази даних або додаткові програмні засобі – все поставляється готовими образами і розгортається в автоматичному режимі.

Висновки до розділу

В розділі було розглянуто створення системи, детально розглянуто створення компоненту, який є мікросервісом, на прикладі Tutors. Обгрунтовано використання підходів у розробці коду застосунку, наведено приклади класів та особливості їх функціонування. Розглянуто взаємодію між мікросервісами в рамках реалізованого застосунку. Описано діаграму послідовності для запиту на надсилання повідомлення користувачу.

ВИСНОВКИ

У результаті виконання дипломного проекту було зроблено Веб-застосунок для пошуку репетитора на базі мікросервісної архітектури.

Проведено аналіз існуючих рішень, обґрунтовано практичну цінність та доцільність розробки даного веб-застосунку. Проведено дослідження сучасних підходів до розробки серверної частини застосунків, проаналізовано та порівняно основні підходи до розробки архітектури серверної частини: монолітної та мікросервісної. Проаналізовано результати дослідження та обрано для проектування системи мікросервісну архітектуру, яка є більш гнучкою і сучасною.

Для мікросервісної архітектури проведено детальне дослідження особливостей проектування та розробки застосунків з використанням різних підходів та технологій. Опрацьовано велику кількістю різноманітних джерел та видань, які стосуються теми мікросервісів та підходів до їх розробки. Надано основні дані про роботу з мікросервісами, розділення та визначення меж функціональності за допомогою аналізу специфікації та особливостей організації, яка потребує розробки системи.

Для більш наочної візуалізації можливостей мікросервісів було використано різні сучасні технологій, що дозволило в процесі розробки познайомитись з різними сторонніми бібліотеками та драйверами.

Розглянуто сучасні методи розгортання застосунків, розроблених з використанням мікросервісної архітектури, на сервері. Проаналізовано основні складнощі при розгортанні, їх причини та можливі варіанти уникнення цих проблем.

В ході роботи в пройдено всі етапи розробки застосунку, від аналізу вимог до створення робочого прототипу, всі необхідні матеріали візуалізовано у вигляді рисунків та схем і додано до пояснювальної записки

Налаштовано систему автоматичного запуску застосунку для персонального комп’ютера за допомогою Docker-хосту. Розроблено прототип проекту для реалізації інтерфейсу користувача та його взаємодії з даними. Представлено функціонал для роботи користувача з системою через інтерфейс.

Розглянуто розробка мікросервісів з використанням .NET Core та протоколу HTTP для налагодження комунікації між ними. Особливістю розробленого проекту є використання Linux-контейнерів в зв’язці з технологіями, традиційно використовуваними на платформі Windows. Це дозволяє запускати проект на будь-якій платформі, яка підтримує базовий функціонал для Docker.

Реалізована система та відповідає меті проекту та задовольняє початкові вимоги.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. github.com [Електронний ресурс] – режим доступу до ресурсу: https://github.com/AlexS98/tutorsOnContainers – Назва з екрану
2. Postman (документація) [Електронний ресурс] – режим доступу: https://www.getpostman.com/collections/1c2d2e6771bab6e75479 – Назва з екрану
3. Docker [Електронний ресурс] – режим доступу: https://towardsdatascience.com/learn-enough-docker-to-be-useful – Назва з екрану
4. Postman (офіційний сайт) [Електронний ресурс] – режим доступу: https://www.getpostman.com/ – Назва з екрану
5. EF Core Data Loading [Електронний ресурс] – режим доступу: https://docs.microsoft.com/en-us/ef/core/querying/related-data – Назва з екрану
6. Demystifying Conway's Law [Електронний ресурс] – режим доступу: https://www.thoughtworks.com/insights/blog/demystifying-conways-law – Назва з екрану
7. Christian Horsdal Gammelgaard «Microservices in .NET Core» / Christian Horsdal Gammelgaard – К. : «Manning», 2017. – 341 с.
8. Monolithic Architecture—Top Design Inspiration Decoration Monolithic Architecture. [Електронний ресурс] – режим доступу: https://medium.com/@putrasulung2108/monolithic-architecture-5dd5961a0688 – Назва з екрану
9. Jon P Smith «Entity Framework Core in Action» / Jon P Smith – К. : «Manning», 2018. – 520 с.
10. Микросервисы — за и против [Електронний ресурс] – режим доступу: http://devopsru.com/news/2016-05-10-microservice-trade-offs.html – Назва з екрану

11. MonolithFirst [Електронний ресурс] – режим доступу: https://martinfowler.com/bliki/MonolithFirst.html – Назва з екрану
12. What is Continuous Delivery [Електронний ресурс] – режим доступу: https://continuousdelivery.com/ – Назва з екрану
13. DRY Programming Practices [Електронний ресурс] – режим доступу: https://metova.com/dry-programming-practices/ – Назва з екрану
14. .NET Core WEB API [Електронний ресурс] – режим доступу: https://metanit.com/sharp/aspnet5/23.1.php – Назва з екрану
15. Entity Framewotk Core [Електронний ресурс] – режим доступу: https://www.learnentityframeworkcore.com/ – Назва з екрану
16. Microsoft SQL Server [Електронний ресурс] – режим доступу: https://en.wikipedia.org/wiki/Microsoft_SQL_Server – Назва з екрану
17. Postgre SQL [Електронний ресурс] – режим доступу: http://www.postgresqltutorial.com/what-is-postgresql/ – Назва з екрану
18. The Redis [Електронний ресурс] – режим доступу: https://redis.io/documentation – Назва з екрану
19. Rabbit MQ [Електронний ресурс] – режим доступу: https://www.rabbitmq.com/getstarted.html – Назва з екрану
20. Model validation in ASP.NET Core [Електронний ресурс] – режим доступу: https://docs.microsoft.com/en-us/aspnet/core/mvc/models/validation – Назва з екрану

ДОДАТОК Г

Код програми